

**Programming in
VAX FORTRAN**

AA-D034D-TE

digital
software

Programming in VAX FORTRAN

AA-D034D-TE

September 1984

This manual is designed to provide all of the basic information needed to use a VAX/VMS V4 system in developing VAX FORTRAN programs. It also provides a complete description of the VAX FORTRAN language.

Software Version: V4.0

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

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


No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright © 1984 by Digital Equipment Corporation
All Rights Reserved.

Printed in U.S.A.

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

The following are trademarks of Digital Equipment Corporation:

DEC	EduSystem	PRO/RMS
DEC/CMS	IAS	PROSE
DEC/MMS	MASSBUS	PROSE PLUS
DECmate	Micro/R SX	Rainbow
DECnet	PDP	RSTS
DECsystem-10	PDT	RSX
DECSYSTEM-20	P/OS	Tool Kit
DECUS	PRO/BASIC	UNIBUS
DECwriter	PRO/Communications	VAX
DIBOL	Professional	VMS
	PRO/FMS	VT
		Work Processor

ZK-2660

HOW TO ORDER ADDITIONAL DOCUMENTATION

In Continental USA and Puerto Rico call 800-258-1710

In New Hampshire, Alaska, and Hawaii call 603-884-6660

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6146 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire 03061

*Any prepaid order from Puerto Rico must be placed
with the local Digital subsidiary (809-754-7575)

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: A&SG Business Manager

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
A&SG Business Manager
c/o Digital's local subsidiary or
approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

Contents

Preface	xix
Chapter 1 Using the VAX/VMS Operating System	Page
1.1 VAX/VMS Commands for Program Development—Overview	1-1
1.1.1 Commands	1-2
1.1.2 Command Qualifiers	1-3
1.2 Accessing the System	1-3
1.2.1 Logging In	1-4
1.2.2 Logging Out	1-4
1.2.3 Changing Your Password	1-5
1.2.4 Declaring Your Terminal Type	1-5
1.2.5 Logging In to Other Network Nodes	1-6
1.3 Using DCL	1-7
1.3.1 Rules for Entering and Editing DCL Commands	1-7
1.3.2 Rules for Defining DCL Symbols	1-8
1.4 Getting HELP From VAX/VMS	1-9
1.5 Working with Files	1-10
1.5.1 File Specifications	1-10
1.5.1.1 File Specification Defaults	1-13
1.5.1.2 Wildcard Characters	1-14
1.5.1.3 Directories and Subdirectories	1-14
1.5.1.4 Logical Names	1-17
1.5.2 File-Handling Commands	1-21
1.5.2.1 Moving Files	1-22
1.5.2.2 Deleting Files	1-23
1.5.2.3 Listing File Names	1-24
1.5.2.4 Renaming Files	1-25
1.5.2.5 Handling File Protections	1-25
1.5.2.6 Searching File Contents	1-26
1.5.2.7 Printing and Typing Files	1-26

1.6	Using Command Procedures	1-27
1.6.1	Using Symbols	1-28
1.6.2	Assigning Character Values to Symbols	1-29
1.6.3	Assigning Numeric Values to Symbols	1-29
1.6.4	Symbol Substitution	1-31
1.6.5	Passing Parameters to Command Procedures.	1-32
1.6.6	Passing Fixed Data to Programs.	1-33
1.6.7	Controlling Command Procedure Input/Output.	1-34
1.6.8	Controlling Command Procedure Execution Flow.	1-35
1.6.9	Handling Command Procedure Errors	1-36
1.6.10	Submitting Command Procedures in Batch Mode	1-37
1.6.11	Login Command File	1-38

Chapter 2 Creating and Modifying Programs Page

2.1	Introduction to EDT	2-1
2.1.1	The Help Facilities	2-2
2.1.2	Invoking and Terminating EDT	2-3
2.1.2.1	Invoking EDT	2-3
2.1.2.2	Terminating an EDT Session	2-4
2.1.3	Entering and Exiting Editing Modes.	2-5
2.1.4	Protecting and Recovering Text	2-5
2.1.5	Creating a New File	2-6
2.2	Character Mode Editing.	2-6
2.2.1	Maneuvering the Cursor	2-7
2.2.2	Inserting New Text	2-10
2.2.3	Deleting and Undeleting Text	2-10
2.2.4	Moving Text	2-11
2.3	Line Mode Editing	2-12
2.3.1	Line Editing Command Summary	2-12
2.3.2	Specifying Line Ranges	2-14
2.3.3	Displaying Lines of Text	2-17
2.3.4	Maneuvering in a File	2-18
2.3.5	Inserting New Text	2-18
2.3.6	Deleting and Replacing Text	2-19
2.3.7	Moving Text	2-20
2.3.8	Substituting Text.	2-20
2.3.9	Input From and Output To Files	2-21
2.3.10	Editing a File From Another Directory	2-22
2.4	EDT Aids for the Programmer.	2-23
2.4.1	Structured Tabs	2-23
2.4.2	Special-Purpose Key Definitions.	2-24
2.4.3	Startup Command Files	2-25

Chapter 3 Compiling FORTRAN Programs Page

3.1	Functions of the Compiler.	3-1
3.2	The FORTRAN Command	3-2

3.2.1	Specifying Input Files	3-2
3.2.2	Specifying Output Files.	3-3
3.2.3	Qualifiers to the FORTRAN Command	3-4
3.2.3.1	/CHECK Qualifier	3-6
3.2.3.2	/CONTINUATIONS Qualifier.	3-7
3.2.3.3	/CROSS_REFERENCE Qualifier.	3-7
3.2.3.4	/DEBUG Qualifier	3-8
3.2.3.5	/D_LINES Qualifier	3-9
3.2.3.6	/DML Qualifier.	3-9
3.2.3.7	/EXTEND_SOURCE Qualifier.	3-9
3.2.3.8	/F77 Qualifier	3-10
3.2.3.9	/G_FLOATING Qualifier.	3-10
3.2.3.10	/I4 Qualifier.	3-11
3.2.3.11	/LIBRARY Qualifier	3-11
3.2.3.12	/LIST Qualifier	3-11
3.2.3.13	/MACHINE_CODE Qualifier	3-12
3.2.3.14	/OBJECT Qualifier	3-12
3.2.3.15	/OPTIMIZE Qualifier	3-12
3.2.3.16	/SHOW Qualifier	3-13
3.2.3.17	/STANDARD Qualifier	3-14
3.2.3.18	/WARNINGS Qualifier	3-15
3.3	Using Text Libraries	3-15
3.3.1	Using the LIBRARY Commands.	3-17
3.3.2	Naming Text Modules	3-17
3.3.3	Specifying Library Files on the FORTRAN Command Line.	3-18
3.3.4	Search Order of Libraries	3-18
3.3.4.1	User-Supplied Default Libraries	3-18
3.3.4.2	System-Supplied Default Library	3-19
3.4	Using the VAX Common Data Dictionary	3-19
3.4.1	Accessing the CDD from FORTRAN Programs.	3-21
3.4.2	Creating CDD Structure Declarations	3-22
3.4.3	FORTRAN and CDDL Data Types	3-22
3.5	Compilation Control Statements.	3-24
3.5.1	INCLUDE Statement.	3-24
3.5.2	OPTIONS Statement.	3-26
3.5.3	DICTIONARY Statement.	3-27
3.6	Compiler Diagnostic Messages and Error Conditions	3-28
3.7	Compiler Output Listing Format	3-28
3.7.1	Source Code Section	3-28
3.7.2	Machine Code Section	3-29
3.7.3	Storage Map Section	3-32
3.7.4	Compilation Summary Section	3-35
Chapter 4 Linking and Running FORTRAN Programs		Page
4.1	Linking FORTRAN Programs	4-1
4.1.1	Functions of the Linker	4-1
4.1.2	The LINK Command.	4-2

4.1.2.1	Linker Output File Qualifiers	4-3
4.1.2.2	/DEBUG and /TRACEBACK Qualifiers	4-5
4.1.2.3	Linker Input File Qualifiers	4-5
4.1.3	Linker Messages	4-6
4.2	Running FORTRAN Programs.	4-6
4.2.1	The RUN Command	4-6
4.2.2	System Processing at Image Exit	4-7
4.2.3	Interrupting a Program	4-7
4.2.4	Returning Status Values to the Command Interpreter	4-8
4.3	Finding and Correcting Run-Time Errors.	4-8
4.3.1	Effects of Error-Related Command Qualifiers	4-9
Chapter 5 Introduction to VAX FORTRAN.		Page
5.1	VAX FORTRAN Language Definition	5-1
5.2	ELEMENTS OF FORTRAN SOURCE PROGRAMS	5-2
5.2.1	Program Units	5-2
5.2.2	Statements.	5-2
5.2.2.1	Order of Statements in a Program Unit	5-2
5.2.3	Symbolic Names	5-4
5.2.4	Comments	5-5
5.3	FORTRAN Character Set.	5-6
5.4	Format Requirements of FORTRAN Source Code	5-7
5.4.1	Fixed-Format Lines.	5-7
5.4.2	Tab-Format Lines	5-9
5.4.3	Statement Label Field	5-10
5.4.3.1	Comment Indicator.	5-10
5.4.3.2	Debugging Statement Indicator	5-10
5.4.4	Continuation Indicator Field	5-11
5.4.5	Statement Field	5-11
5.4.6	Sequence Number Field.	5-11
Chapter 6 Data Types, Data Items, and Expressions		Page
6.1	Data Types.	6-1
6.1.1	Storage Requirements.	6-2
6.1.2	VAX Implementations of REAL*8.	6-3
6.2	Data Items	6-4
6.2.1	Constants	6-4
6.2.1.1	Integer Constants.	6-5
6.2.1.2	Real Constants	6-5
6.2.1.3	Complex Constants	6-9
6.2.1.4	Octal and Hexadecimal Constants.	6-11
6.2.1.5	Logical Constants.	6-13
6.2.1.6	Character Constants	6-13
6.2.1.7	Hollerith Constants.	6-14

6.2.2	Variables	6-15
6.2.2.1	Data Type by Specification	6-16
6.2.2.2	Data Type by Implication	6-17
6.2.3	Arrays	6-17
6.2.3.1	Array Declarators	6-18
6.2.3.2	Array Subscripts	6-19
6.2.3.3	Arrangement of Array Elements in Storage	6-19
6.2.3.4	Data Type of an Array	6-21
6.2.3.5	Array References Without Subscripts	6-21
6.2.3.6	Adjustable Arrays	6-21
6.2.3.7	Assumed-Size Arrays	6-21
6.2.4	Character Substrings	6-22
6.2.5	Records	6-23
6.2.5.1	Overview of Records and Structures	6-23
6.2.5.2	Arrangement of Records in Storage	6-24
6.2.5.3	Record and Field References	6-30
6.2.6	Terminology Used to Refer to Data Items	6-31
6.3	Expressions	6-33
6.3.1	Arithmetic Expressions	6-33
6.3.1.1	Use of Parentheses	6-34
6.3.1.2	Data Type of an Arithmetic Expression	6-35
6.3.2	Character Expressions	6-37
6.3.3	Relational Expressions	6-38
6.3.4	Logical Expressions	6-39
Chapter 7 Assignment Statements		Page
7.1	Arithmetic Assignment Statement	7-1
7.2	Logical Assignment Statement	7-3
7.3	Character Assignment Statement	7-4
7.4	Aggregate Assignment Statement	7-5
7.5	ASSIGN Statement	7-6
Chapter 8 Specification Statements		Page
8.1	BLOCK DATA Statement	8-2
8.2	COMMON Statement	8-3
8.3	DATA Statement	8-4
8.4	Data Type Declaration Statements	8-7
8.4.1	Numeric Type Declaration Statements	8-7
8.4.2	Character Type Declaration Statements	8-8
8.5	DIMENSION Statement	8-9
8.6	EQUIVALENCE Statement	8-10
8.6.1	Making Arrays Equivalent	8-11
8.6.2	Making Substrings Equivalent	8-13
8.6.3	EQUIVALENCE and COMMON Interaction	8-16

8.7	EXTERNAL Statement	8-16
8.8	IMPLICIT Statement	8-17
8.9	INTRINSIC Statement	8-18
8.10	NAMelist Statement	8-20
8.11	PARAMETER Statement	8-21
8.12	PROGRAM Statement	8-23
8.13	RECORD Statement	8-23
8.14	SAVE Statement	8-24
8.15	Structure Declaration Block	8-25
	8.15.1 Structure Declaration	8-26
	8.15.2 Substructure Declarations	8-31
	8.15.3 Union Declarations	8-31
8.16	VOLATILE Statement	8-34

Chapter 9 CONTROL Statements Page

9.1	CALL Statement	9-2
9.2	CONTINUE Statement	9-3
9.3	DO Statement	9-3
9.3.1	Indexed DO Statement	9-3
	9.3.1.1 DO Iteration Control	9-4
	9.3.1.2 Nested DO Loops	9-6
	9.3.1.3 Control Transfers in DO Loops	9-7
	9.3.1.4 Extended Range	9-7
9.3.2	DO WHILE Statement	9-9
9.4	END DO Statement	9-9
9.5	END Statement	9-10
9.6	GO TO Statements	9-10
	9.6.1 Unconditional GO TO Statement	9-10
	9.6.2 Computed GO TO Statement	9-11
	9.6.3 Assigned GO TO Statement	9-12
9.7	IF Statements	9-12
	9.7.1 Arithmetic IF Statement	9-13
	9.7.2 Logical IF Statement	9-13
	9.7.3 Block IF Statements	9-14
	9.7.3.1 Statement Blocks	9-17
	9.7.3.2 Block IF Examples	9-17
	9.7.3.3 Nested Block IF Constructs	9-19
9.8	PAUSE Statement	9-20
9.9	RETURN Statement	9-21
9.10	STOP Statement	9-23

Chapter 10 Subroutines and Functions — Subprograms Page

10.1	Subprogram Arguments	10-2
10.1.1	Actual Argument and Dummy Argument Association	10-2
10.1.1.1	Adjustable Arrays	10-3
10.1.1.2	Assumed-Size Arrays	10-4

10.1.1.3	Passed-Length Character Arguments	10-5
10.1.1.4	Character and Hollerith Constants as Actual Arguments	10-6
10.1.1.5	Alternate Return Arguments.	10-6
10.1.2	Built-In Functions	10-7
10.1.2.1	Argument List Built-In Functions	10-7
10.1.2.2	%LOC Built-In Function	10-8
10.2	User-Written Subprograms	10-9
10.2.1	Statement Functions	10-9
10.2.2	Function Subprograms	10-11
10.2.2.1	Logical and Numeric Functions	10-12
10.2.2.2	Character Functions	10-12
10.2.2.3	Function Reference	10-13
10.2.3	Subroutine Subprograms — SUBROUTINE Statement.	10-14
10.2.4	ENTRY Statement	10-16
10.2.4.1	ENTRY in Function Subprograms	10-17
10.2.4.2	ENTRY in Subroutine Subprograms	10-18
10.3	FORTRAN Intrinsic Functions	10-19
10.3.1	Intrinsic Function References	10-19
10.3.2	Generic Function References	10-20
10.3.3	Intrinsic and Generic Function Usage	10-22
10.3.4	Character and Lexical Comparison Library Functions	10-24
10.3.4.1	Character Functions	10-24
10.3.4.2	Lexical Comparison Functions	10-26
Chapter 11 VAX FORTRAN Input/Output		Page
11.1	Overview of VAX FORTRAN I/O.	11-1
11.1.1	Identifying Logical Input/Output Units	11-2
11.1.2	Types of I/O Statements	11-2
11.1.3	Interprocess Communication	11-3
11.1.4	Forms of I/O Statements	11-3
11.2	Elements of I/O Processing	11-4
11.2.1	File Specifications	11-4
11.2.2	Logical Names and Logical Unit Numbers.	11-5
11.2.2.1	FORTRAN Logical Names.	11-6
11.2.2.2	Implied FORTRAN Logical Unit Numbers	11-7
11.2.2.3	File Specification in the OPEN Statement	11-8
11.2.2.4	Assigning Files to Logical Units—Summary.	11-9
11.2.3	File Organizations, I/O Record Formats, and Access Modes.	11-10
11.2.3.1	File Organizations	11-10
11.2.3.2	Internal Files	11-11
11.2.3.3	I/O Record Formats	11-12
11.2.3.4	Record Access Modes	11-14

11.3	Components of I/O Statements	11-16
11.3.1	Control List	11-16
11.3.1.1	Logical Unit Specifier	11-17
11.3.1.2	Internal File Specifier	11-17
11.3.1.3	Format Specifiers	11-17
11.3.1.4	Namelist Specifier	11-18
11.3.1.5	Record Specifier	11-19
11.3.1.6	Key-Field-Value Specifier	11-19
11.3.1.7	Key-of-Reference Specifier	11-20
11.3.1.8	I/O Status Specifier	11-21
11.3.1.9	Transfer-of-Control Specifiers	11-21
11.3.1.10	Rules for Specifying Control List Parameters—Summary	11-22
11.3.2	I/O List	11-23
11.3.2.1	Simple List Elements	11-23
11.3.2.2	Implied-DO Lists in I/O Statements	11-24
11.4	READ Statements	11-26
11.4.1	Sequential READ Statements	11-26
11.4.1.1	Formatted Sequential READ Statement	11-27
11.4.1.2	List-Directed Sequential READ Statement	11-28
11.4.1.3	Namelist-Directed Sequential READ Statement	11-30
11.4.1.4	Unformatted Sequential READ Statement	11-34
11.4.2	Direct Access READ Statements	11-35
11.4.2.1	Formatted Direct Access READ Statement	11-36
11.4.2.2	Unformatted Direct Access READ Statement	11-36
11.4.3	Indexed READ Statements	11-37
11.4.3.1	Formatted Indexed READ Statement	11-38
11.4.3.2	Unformatted Indexed READ Statement	11-38
11.4.4	Internal READ Statement	11-39
11.4.4.1	Formatted Internal READ Statement	11-40
11.4.4.2	List-Directed Internal READ Statement	11-40
11.5	WRITE Statements	11-41
11.5.1	Sequential WRITE Statements	11-41
11.5.1.1	Formatted Sequential WRITE Statement	11-42
11.5.1.2	List-Directed Sequential WRITE Statement	11-43
11.5.1.3	Namelist-Directed Sequential WRITE Statement	11-44
11.5.1.4	Unformatted Sequential WRITE Statement	11-45
11.5.2	Direct Access WRITE Statements	11-46
11.5.2.1	Formatted Direct Access WRITE Statement	11-47
11.5.2.2	Unformatted Direct Access WRITE Statement	11-47
11.5.3	Indexed WRITE Statements	11-47
11.5.3.1	Formatted Indexed WRITE Statement	11-48
11.5.3.2	Unformatted Indexed WRITE Statement	11-49

11.5.4	Internal WRITE Statement	11-49
11.5.4.1	Formatted Internal WRITE Statement	11-50
11.5.4.2	List-Directed Internal WRITE Statement	11-50
11.6	REWRITE Statement	11-50
11.6.1	Formatted REWRITE Statement	11-51
11.6.2	Unformatted REWRITE Statement	11-51
11.7	ACCEPT Statement	11-52
11.8	TYPE and PRINT Statements	11-53

Chapter 12	Format Statements.	Page
12.1	Syntax of Format Statement	12-1
12.2	Field and Edit Descriptors	12-3
12.2.1	Repeat Counts and Group Repeat Counts	12-3
12.2.2	Variable Format Expressions	12-4
12.2.3	Blank Control Editing	12-5
12.2.3.1	BN Edit Descriptor	12-5
12.2.3.2	BZ Edit Descriptor	12-5
12.2.4	Sign Control Editing	12-6
12.2.4.1	SP Edit Descriptor	12-6
12.2.4.2	SS Edit Descriptor	12-6
12.2.4.3	S Edit Descriptor	12-6
12.2.5	Integer Editing	12-6
12.2.5.1	I Field Descriptor	12-6
12.2.5.2	O Field Descriptor	12-8
12.2.5.3	Z Field Descriptor	12-9
12.2.6	Real Editing	12-10
12.2.6.1	F Field Descriptor	12-10
12.2.6.2	E Field Descriptor	12-11
12.2.6.3	D Field Descriptor	12-13
12.2.6.4	G Field Descriptor	12-13
12.2.6.5	Complex Data Editing	12-15
12.2.7	Scale Factor Editing — P Edit Descriptor	12-16
12.2.8	Logical Editing — L Edit Descriptor	12-18
12.2.9	Character Editing	12-18
12.2.9.1	A Field Descriptor	12-18
12.2.9.2	H Field Descriptor	12-20
12.2.9.3	Character Constants	12-20
12.2.10	Default Field Descriptors	12-21
12.2.11	Positional Editing	12-22
12.2.11.1	X Edit Descriptor	12-22
12.2.11.2	T Edit Descriptor	12-22
12.2.11.3	TL Edit Descriptor	12-23
12.2.11.4	TR Edit Descriptor	12-23

12.2.12	Miscellaneous Editing Operations	12-24
12.2.12.1	Q Edit Descriptor	12-24
12.2.12.2	Dollar Sign Descriptor	12-24
12.2.12.3	Colon Descriptor	12-25
12.3	Carriage Control	12-25
12.4	Format Specification Separators	12-26
12.5	External Field Separators	12-27
12.6	Run-Time Format	12-27
12.7	Format Control Interaction With I/O Lists	12-28
12.8	Summary of Rules for Format Statements	12-29
12.8.1	General Rules	12-31
12.8.2	Input Rules	12-32
12.8.3	Output Rules.	12-32

Chapter 13 Auxiliary Input/Output Statements Page

13.1	OPEN Statement	13-1
13.1.1	ACCESS Keyword	13-6
13.1.2	ASSOCIATEVARIABLE Keyword.	13-6
13.1.3	BLANK Keyword	13-6
13.1.4	BLOCKSIZE Keyword	13-7
13.1.5	BUFFERCOUNT Keyword	13-7
13.1.6	CARRIAGECONTROL Keyword	13-8
13.1.7	DEFAULTFILE Keyword.	13-8
13.1.8	DISPOSE Keyword.	13-9
13.1.9	ERR Keyword	13-9
13.1.10	EXTENDSIZE Keyword.	13-10
13.1.11	FILE Keyword	13-10
13.1.12	FORM Keyword.	13-10
13.1.13	INITIALSIZE Keyword	13-11
13.1.14	IOSTAT Keyword	13-11
13.1.15	KEY Keyword	13-12
13.1.16	MAXREC Keyword	13-13
13.1.17	NAME Keyword	13-13
13.1.18	NOSPANBLOCKS Keyword.	13-13
13.1.19	ORGANIZATION Keyword	13-13
13.1.20	READONLY Keyword.	13-14
13.1.21	RECL Keyword	13-14
13.1.22	RECORDSIZE Keyword	13-15
13.1.23	RECORDTYPE Keyword	13-15
13.1.24	SHARED Keyword	13-16
13.1.25	STATUS Keyword	13-16
13.1.26	TYPE Keyword	13-17
13.1.27	UNIT Keyword	13-17
13.1.28	USEROPEN Keyword.	13-17
13.2	CLOSE Statement.	13-18
13.3	INQUIRE Statement.	13-19
13.3.1	ACCESS Specifier	13-20
13.3.2	BLANK Specifier.	13-20
13.3.3	CARRIAGECONTROL Specifier	13-20
13.3.4	DIRECT Specifier	13-21

13.3.5	ERR Specifier	13-21
13.3.6	EXIST Specifier	13-21
13.3.7	FORM Specifier	13-22
13.3.8	FORMATTED Specifier	13-22
13.3.9	IOSTAT Specifier	13-22
13.3.10	KEYED Specifier	13-22
13.3.11	NAME Specifier	13-23
13.3.12	NAMED Specifier	13-23
13.3.13	NEXTREC Specifier	13-24
13.3.14	NUMBER Specifier	13-24
13.3.15	OPENED Specifier	13-24
13.3.16	ORGANIZATION Specifier	13-24
13.3.17	RECL Specifier	13-25
13.3.18	RECORDTYPE Specifier	13-25
13.3.19	SEQUENTIAL Specifier	13-25
13.3.20	UNFORMATTED Specifier	13-26
13.4	REWIND Statement	13-26
13.5	BACKSPACE Statement	13-27
13.6	ENDFILE Statement	13-27
13.7	DELETE Statement	13-28
13.8	UNLOCK Statement	13-30

Chapter 14 Using Structures and Records Page

14.1	Structures	14-2
14.2	Records	14-3
14.3	Uses of Records	14-3

Chapter 15 Using Indexed Files

15.1	Creating an Indexed File	15-2
15.2	Writing Indexed Files	15-3
15.2.1	Duplicate Values in Key Fields	15-3
15.2.2	Preventing the Indexing of Alternate Key Fields	15-4
15.3	Reading Indexed Files	15-5
15.4	Updating Records	15-6
15.5	Deleting Records	15-7
15.6	Current Record and Next Record Pointers	15-7
15.7	Exception Conditions	15-7

Chapter 16 Using Character Data Page

16.1	Character Substrings	16-1
16.2	Building Character Strings	16-2
16.3	Character Constants	16-2
16.4	Declaring Character Data	16-3
16.5	Initializing Character Variables	16-4
16.6	Passed-Length Character Arguments	16-4
16.7	Character Library Functions	16-5
16.7.1	CHAR Function	16-5
16.7.2	ICHAR Function	16-6
16.7.3	INDEX Function	16-6
16.7.4	LEN Function	16-6
16.7.5	LGE, LGT, LLE, LLT Functions	16-7

16.8	Character Data Examples	16-7
16.9	Character I/O	16-7

Chapter 17 Debugging VAX FORTRAN Programs Page

17.1	Debugging Overview	17-2
17.2	Preparing a Program for Debugging—Compiling and Linking.	17-2
17.3	Invoking and Terminating the Debugger	17-3
17.3.1	Invoking the Debugger with the RUN Command	17-3
17.3.2	Invoking the Debugger During Program Execution	17-4
17.3.3	Suspending the Debugger to Issue DCL Commands	17-4
17.3.4	Interrupting the Debugger	17-5
17.3.5	Terminating a Debugger Session	17-5
17.4	The Debugger Environment	17-5
17.4.1	Using Debugger HELP	17-5
17.4.2	Entering Commands	17-6
17.4.2.1	Normal Keyboard Entry	17-6
17.4.2.2	Keypad Entry	17-6
17.4.3	User-Defined Keypad Command Keys	17-7
17.4.4	Using Debugger Command Procedures.	17-8
17.4.5	Initializing a Debugging Session.	17-9
17.4.6	Recording Debug Sessions in Log Files.	17-9
17.4.7	Debugger Command Syntax and Summary	17-10
17.5	Controlling Program Execution	17-17
17.5.1	Starting Program Execution.	17-17
17.5.1.1	GO Command.	17-17
17.5.1.2	STEP Command	17-18
17.5.1.3	CALL Command	17-19
17.5.1.4	SHOW CALLS Command	17-19
17.5.2	Suspending or Tracing Program Execution.	17-20
17.5.2.1	Breakpoints and Tracepoints.	17-20
17.5.2.2	Watchpoints	17-23
17.5.3	Displaying Source Lines	17-23
17.5.4	Using the Debugger's Logical Control Commands	17-25
17.6	Using Symbolic Names and Accessing Program Locations	17-25
17.6.1	Making Symbolic Names Accessible — SET MODULE	17-26
17.6.2	Referencing Locations in a Program	17-28
17.6.2.1	Specifying Data Addresses	17-28
17.6.2.2	Current, Previous, and Next Locations	17-28
17.6.2.3	Specifying Program Addresses	17-29
17.6.3	Making Symbolic References Unique — Prefixes and Scope.	17-29
17.6.3.1	Pathname Prefix	17-30
17.6.3.2	SET SCOPE Command	17-31
17.6.4	Defining Addresses Symbolically	17-31
17.6.5	Displaying Symbol Information — SHOW SYMBOL.	17-31

17.7	Examining and Manipulating Data	17-32
17.7.1	Hints about the Use of Expressions	17-33
17.7.2	Displaying Values — EXAMINE	17-34
17.7.3	Calculating Values — EVALUATE	17-35
17.7.4	Assigning Values — DEPOSIT	17-35
17.7.5	Specifying Data Type.	17-35
17.7.6	Specifying Radix	17-36
17.7.7	Using Numeric Data Types in Expressions.	17-37
17.8	Using Screen Displays	17-38
17.8.1	Invoking and Terminating Screen Mode	17-39
17.8.2	Defining Windows	17-40
17.8.3	Manipulating Displays	17-41
17.8.3.1	Scrolling Screen Displays	17-41
17.8.3.2	Creating Screen Displays	17-41
17.8.3.3	Accessing Displays.	17-43
17.8.3.4	Removing Screen Displays	17-43
17.9	Sample Debugging Sessions	17-43
17.9.1	Debugging a FORTRAN Program Unit	17-43
17.9.2	Debugging a FORTRAN Program with Subprograms	17-46

Chapter 18 Error Processing. Page

18.1	Run-Time Library Default Error Processing	18-1
18.2	Using the ERR and END Specifiers.	18-6
18.3	Using the IOSTAT Specifier	18-6

Appendix A Additional Language Elements Page

A.1	The ENCODE and DECODE Statements	A-1
A.2	DEFINE FILE Statement	A-3
A.3	FIND Statement	A-4
A.4	PARAMETER Statement.	A-5
A.5	Octal Notation for Integer Constants	A-6
A.6	/NOF77 Interpretation of the EXTERNAL Statement	A-6

Appendix B Character Sets Page

B.1	FORTRAN Character Set.	B-1
B.2	ASCII Character Set	B-2
B.3	Radix-50 Constants and Character Set	B-3

Appendix C FORTRAN Data Representation Page

C.1	LOGICAL*1 (Byte) Representation	C-1
C.2	INTEGER*2 Representation	C-1
C.3	INTEGER*4 Representation	C-2
C.4	Floating-Point Representations	C-2
C.4.1	REAL*4 (F__floating)	C-2
C.4.2	REAL*8 (D__floating)	C-3
C.4.3	REAL*8 (G__floating)	C-4
C.4.4	REAL*16 (H__floating)	C-4
C.4.5	COMPLEX*8 (F__floating)	C-6

C.4.6	COMPLEX*16 (D__floating)	C-6
C.4.7	COMPLEX*16 (G__floating)	C-7
C.5	Logical Representation	C-8
C.6	Character Representation	C-10
C.7	Hollerith Representation	C-10

Appendix D FORTRAN Language Summary Page

D.1	Expression Operators	D-1
D.2	Statements	D-2
D.3	LIBRARY Functions	D-30
D.4	System Subroutine Summary	D-38
D.4.1	DATE Subroutine	D-39
D.4.2	IDATE Subroutine	D-39
D.4.3	ERRSNS Subroutine	D-40
D.4.4	EXIT Subroutine	D-40
D.4.5	SECNDS Subroutine	D-41
D.4.6	TIME Subroutine	D-41
D.4.7	RAN Subroutine	D-42
D.4.8	MVBITS Subroutine	D-42
D.5	Bit Functions	D-43
D.5.1	Bit Position	D-43
D.5.2	Bit Function Arguments	D-43

Appendix E Diagnostic Messages Page

E.1	Diagnostic Messages From the Compiler	E-1
E.1.1	Source Program Diagnostic Messages	E-1
E.1.2	Compiler-Fatal Diagnostic Messages	E-26
E.1.3	Compiler Limits	E-27
E.2	Run-Time Diagnostic Messages	E-28
E.2.1	Run-Time Library Diagnostic Message Presentation	E-28
E.2.2	Run-Time Library Diagnostic Messages	E-29
E.3	Dictionary Error Messages	E-41

Figures Page

1-1	Program Development Process	1-2
1-2	A Directory Hierarchy	1-16
2-1	VT52 Keypad	2-8
2-2	VT100 Keypad	2-8
2-3	VT200 Keypad	2-9
3-1	Creating and Using a Text Library	3-16
3-2	Sample Listing of Source Code	3-29
3-3	Sample Listing of Machine Code	3-30
3-4	Sample Storage Map Section	3-34
3-5	Sample Compilation Summary	3-35
4-1	Sample FORTRAN Program and Traceback	4-10
5-1	Required Order of Statements and Lines	5-3
5-2	FORTTRAN Coding Form	5-8
5-3	Line Formatting Example	5-9

6-1	Array Storage	6-20
8-1	Equivalence of Array Storage	8-12
8-2	Equivalence of Arrays with Nonunity Lower Bounds	8-13
8-3	Equivalence of Substrings.	8-14
8-4	Equivalence of Character Arrays	8-15
9-1	Nested DO Loops	9-6
9-2	Control Transfers and Extended Range	9-8
9-3	Examples of Block IF Constructs	9-16
10-1	Multiple Functions in a Function Subprogram	10-18
10-2	Multiple Function Name Usage	10-22
12-1	Variable Format Expression Example	12-5
16-1	Character Data Program Example	16-9
16-2	Output Generated by Example Program	16-10
17-1	Default Keypad Definitions—VT100	17-7
17-2	Sample FORTRAN Program	17-43
17-3	Sample Debugging Session.	17-44
17-4	Sample FORTRAN Program with Subprograms.	17-46
17-5	Sample Multiunit Debugging Session.	17-47
E-1	Sample Diagnostic Messages (Listing Format).	E-3

Tables	Page	
1-1	Common File Types	1-12
1-2	File Specification Defaults	1-13
1-3	Predefined System Logical Names.	1-19
1-4	Summary of Operators in Expressions	1-30
1-5	Severity Codes	1-36
2-1	Summary of Line Editing Commands	2-13
2-2	Single-Line Range Specifications	2-15
2-3	Multiple-Line Range Specifications	2-16
3-1	FORTRAN Command Qualifiers	3-5
3-2	Commands to Control Library Files	3-17
4-1	LINK Command Qualifiers	4-2
4-2	/DEBUG and /TRACEBACK Qualifiers	4-9
5-1	Entities Identified by Symbolic Names	5-5
6-1	Data Type Storage Requirements	6-2
7-1	Conversion Rules for Assignment Statements	7-3
10-1	Argument List Built-In Functions and Defaults	10-8
10-2	Types of User-Written Subprogram.	10-9
10-3	Generic Function Name Summary	10-21
11-1	Available I/O Statements	11-4
11-2	Predefined System Logical Names	11-6
11-3	Implicit FORTRAN Logical Units	11-8
11-4	Valid Combinations of Record Access Mode and File Organization.	11-14
11-5	List-Directed Output Formats	11-43
12-1	Effect of Data Magnitude on G Format Conversions.	12-14
12-2	Default Field Descriptor Values	12-21
12-3	Carriage Control Characters	12-25
12-4	Summary of FORMAT Codes	12-30
13-1	OPEN Statement Keyword Values	13-3
13-2	Record Size (RECL) Limits	13-14
17-1	Summary of Debug Commands	17-11
17-2	Debugger Command Qualifiers.	17-33
17-3	Debugger Operators	17-34

18-1	Summary of FORTRAN Run-Time Errors	18-3
B-1	ASCII Character Set	B-2
D-1	Expression Operators.	D-1
D-2	VAX FORTRAN Statements	D-2
D-3	Intrinsic Functions.	D-31
E-1	Source Program Diagnostic Messages	E-4
E-2	Compiler-Fatal Diagnostic Messages	E-27
E-3	Compiler Limits	E-28
E-4	Run-Time Diagnostic Messages	E-29
E-5	CRX Error Messages	E-41

Preface

Manual Objectives

The primary objective of this document is to present a complete description of the elements of DIGITAL's VAX FORTRAN language and to explain how to create, compile, link, execute, and debug FORTRAN programs on the VAX/VMS operating system.

This manual is designed to serve as a reference document, not as a tutorial document.

Detailed information about how to optimize VAX FORTRAN programs and how to access VAX/VMS system services is provided, for advanced programmers, in the *VAX FORTRAN User's Guide*.

Intended Audience

This manual is intended for programmers and students who have a basic understanding of the FORTRAN language. It is not necessary for the reader to have a detailed understanding of the VAX/VMS operating system, but some familiarity with VAX/VMS is helpful. For detailed information concerning VAX/VMS, refer to the documents listed under the heading "Associated Documents."

Structure of This Document

To promote ease of reference, this manual is divided into four major segments:

- Section I — Chapters 1 through 4
Provides general information about how to use the VAX/VMS system; how to create a source file; and how to compile, link, and run a program in a source file.
- Section II — Chapters 5 through 13
Provides a complete specification of the VAX FORTRAN language.

- Section III — Chapters 14 through 18

Provides application-oriented information about several important VAX FORTRAN extensions to standard FORTRAN (that is, structures and records, indexed files, and character data) and provides information on the VAX/VMS Symbolic Debugger and on error handling.

- Section IV — Appendixes A through E

Contains information on some additional statements and language features that provide compatible support for programs written in older versions of FORTRAN; summarizes the character sets supported by VAX FORTRAN; describes how VAX FORTRAN data is represented in memory; summarizes the language elements and intrinsic functions supported by VAX FORTRAN; and lists compilation and run-time messages.

Associated Documents

The following documents may be of interest:

- *Introduction to VAX/VMS*
- *VAX FORTRAN User's Guide*
- *Guide to Using DCL and Command Procedures on VAX/VMS*
- *VAX/VMS Run-Time Library Routines Reference Manual*
- *VAX/VMS Symbolic Debugger Reference Manual*
- *VAX/VMS Linker Reference Manual*

For a complete list of all VAX documents, including brief descriptions of each, see the *VAX/VMS Master Index*.

Conventions Used in This Document

The following syntactic conventions are used in this manual:

- Uppercase words and letters used in examples indicate that you should type the word or letter as shown.
- Lowercase words and letters used in syntax specifications indicate that you are to substitute a word or value of your choice.
- Brackets ([]) indicate optional elements.
- Braces ({}) are used to enclose lists from which one element is to be chosen.
- Ellipses (...) indicate that the preceding item(s) can be repeated one or more times.

- “Real” (lowercase) is used to refer to the REAL*4 (REAL), REAL*8, and REAL*16 data types as a group; likewise, “complex” (lowercase) is used to refer to the COMPLEX*8 (COMPLEX) and COMPLEX*16 (DOUBLE COMPLEX) data types as a group; “logical” (lowercase) is used to refer to the LOGICAL*2 and LOGICAL*4 data types as a group; and “integer” (lowercase) is used to refer to the INTEGER*2 and INTEGER*4 data types as a group.
- VAX FORTRAN extensions to the FORTRAN-77 standard are printed in blue.

In addition, the following notations are used to denote special nonprinting characters:

Tab character <TAB>

Space character △

Chapter 1

Using the VAX/VMS Operating System

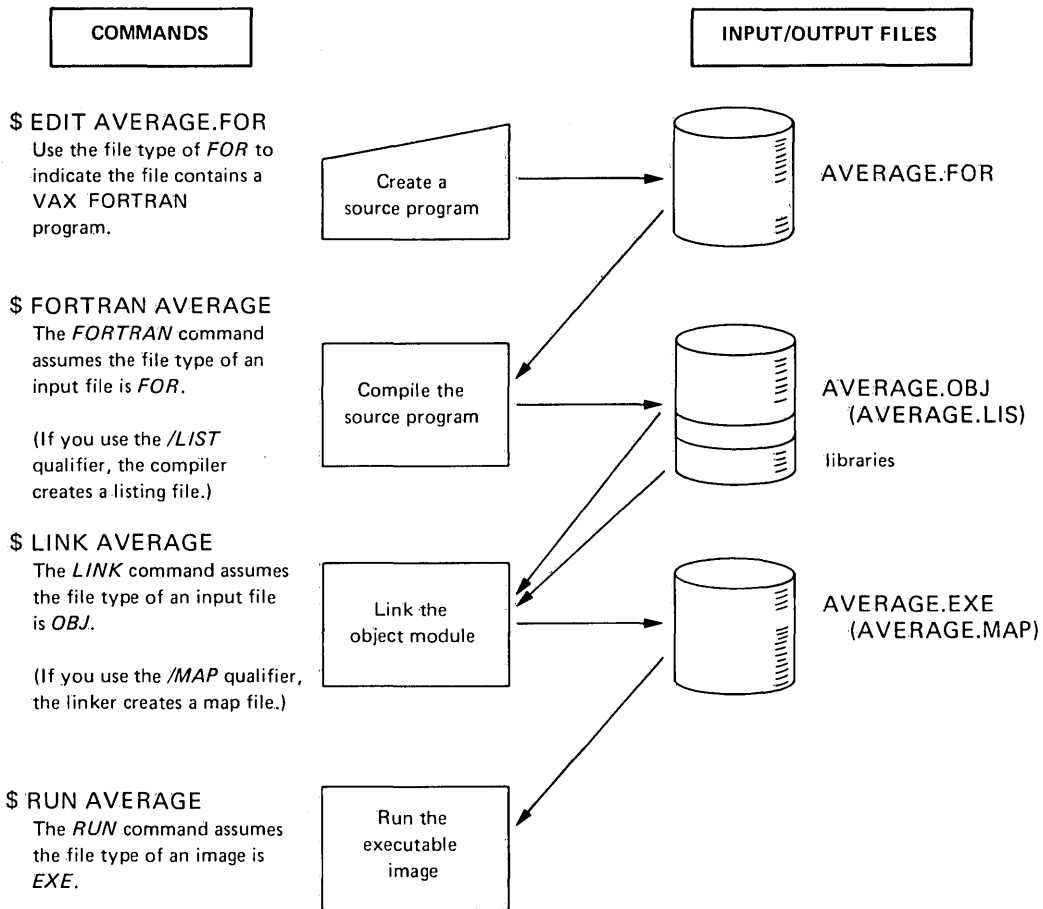
The VAX/VMS operating system and the Digital Command Language (DCL) provide numerous tools and utilities for program development. This chapter summarizes what you need to know to use a VAX/VMS system. The following topics are discussed in this chapter:

- How to create and execute a FORTRAN program—an overview
- How to access a VAX/VMS system
- How to enter DCL commands
- How to get help from VAX/VMS
- How to handle files
- How to use command procedures

Whenever possible, the system information presented in this chapter is oriented toward FORTRAN. A tutorial presentation of the system-specific information is provided in the *Introduction to VAX/VMS*. The *Guide to Using DCL and Command Procedures on VAX/VMS* gives detailed definitions of commands and file specifications.

1.1 VAX/VMS Commands for Program Development—OVERVIEW

Figure 1-1 illustrates the development of a FORTRAN program from its inception to its execution.



ZK-791-82

Figure 1-1: Program Development Process

1.1.1 Commands

The steps shown in Figure 1-1 are performed when you enter the following commands to the VAX system:

```
$ EDIT file-spec
$ FORTRAN file-spec
$ LINK file-spec
$ RUN file-spec
```

(Note: File-spec is the file specification of the file to be processed.)

The **EDIT** command and the EDT editor are described in detail in Chapter 2, the **FORTTRAN** command is described in Section 3.2, and the **LINK** and **RUN** commands are described in Sections 4.1 and 4.2, respectively.

Qualifiers can be included with each command to define further system actions or to modify the processing performed by the system.

1.1.2 Command Qualifiers

Qualifiers specify special actions to be performed. They can be used on the `OPTIONS` statement and on `FORTTRAN`, `LINK`, and `RUN` commands. Qualifiers have the following form:

```
/qualifier
```

Many qualifiers have a corresponding negative form, `/NO`qualifier, which negates the specified action. For example, the `/LIST` qualifier specifies that the compiler is to produce a listing file. The `/NOLIST` qualifier specifies that the compiler is not to produce a listing file.

Some qualifiers accept values, allowing you to activate or deactivate a particular form of processing. To specify a qualifier value, type the qualifier name followed by an equal sign (=) and the value. For example:

```
/CHECK=BOUNDS
```

To specify a list of qualifier values, enclose the values in parentheses. For example:

```
/CHECK=(BOUNDS,OVERFLOW)
```

Defaults have been established for each qualifier, based on the actions that are appropriate in most cases. Sections 3.2.3 and 4.1.2, which describe the qualifiers for each command, contain tables indicating the defaults.

You can specify qualifiers so that all files included in the command are affected, or only certain files are affected. When the qualifier follows the command name immediately, it applies to all files. For example, if you specify the following command, you receive listing files for `ABC`, `XYZ`, and `RST`.

```
$ FORTTRAN/LIST ABC,XYZ,RST
```

If you include a qualifier as part of a file specification, in most cases it affects only the file with which it is associated. For example, use of the `/NOLIST` qualifier in the following command provides you with listing files for `ABC` and `RST`, but not for `XYZ`.

```
$ FORTTRAN/LIST ABC,XYZ/NOLIST,RST
```

Qualifiers included with file specifications that are part of a concatenated list of input files are exceptions to this rule. See Example 5 in Section 3.2.2.

1.2 Accessing the System

In order to develop VAX FORTRAN programs, you must be able to log in and out of the VAX/VMS system. During the login process, the system determines whether your account is valid and your password is correct. The logout process releases the system resources assigned to you and protects your account.

Sections 1.2.1 through 1.2.5 describe the procedures you use to do the following:

- Log in and out
- Change your password
- Give the system information about the kind of terminal you have and its data transmission rate
- Connect your terminal to another network node, that is, another computer in a network that includes your computer

1.2.1 Logging In

Accessing a VAX/VMS system from a terminal is very simple. Once you have turned on the terminal, you alert the system to your presence by typing RETURN, <CTRL/C>, or <CTRL/Y>. Then, VAX/VMS issues the following prompt:

```
Username:
```

You respond by typing in your account name and a RETURN. (Note: Some VAX/VMS systems include an optional feature that requires you to enter a system password before it issues the "Username:" prompt.)

Next, VAX/VMS prompts you with the following:

```
Password:
```

You respond by typing in your password and a RETURN. Note that VAX/VMS does not display your password when you type it on the terminal. This helps to protect your password from accidental discovery.

The following example shows the login procedure:

```
<RET>
```

```
Username: JONES
```

```
Password:
```

```
        Welcome to VAX/VMS Version 4.0 on node BACKUS
```

1.2.2 Logging Out

To end your terminal session, you type the LOGOUT command. VAX/VMS responds with the brief form of the logout message, as shown in the following example:

```
$ LOGOUT
  JONES          logged out at 30-MAR-1984 12:30:00.00
```

If you add the /FULL qualifier to the LOGOUT command (or if VAX/VMS executes the LOGOUT command in a batch file), the long form of the logout message is printed. For example:

```
$ LOGOUT/FULL
  JONES          logged out at 30-MAR-1984 12:30:00.00
```

```
Accounting information:
Buffered I/O count:      22      Peak working set size:  90
Direct I/O count:       10      Peak virtual size:     69
Page faults:            68      Mounted volumes:        0
Elapsed CPU time: 0 00:01:30.50 Elapsed time:  0 04:59:02.63
```

1.2.3 Changing Your Password

If your system manager allows it, you can change your password with the SET PASSWORD command. (Note that you must use this command interactively from your terminal; it cannot be executed in a batch file.) The steps for changing your password are as follows:

1. You type SET PASSWORD, followed by a RETURN. VAX/VMS then prompts you to enter your current password.
2. Enter your current password, followed by a RETURN. Note that VAX/VMS does not display the current password. VAX/VMS then prompts you for your new password, which can have a maximum of 31 characters chosen from the set A through Z, a through z, 0 through 9, dollar sign (\$), and underscore (_). Note that VAX/VMS does not display the new password.
3. To verify that you have typed your new password correctly, VAX/VMS prompts you to enter your new password again. If the two new passwords do not match, the original password remains in effect.

A sample dialog follows:

```
$ SET PASSWORD
Old Password:
New Password:
Verification:
$
```

1.2.4 Declaring Your Terminal Type

When you log in, VAX/VMS may or may not know what type of terminal you have. To set the terminal type, you use the SET TERMINAL command. The simplest form is:

```
SET TERMINAL/INQUIRE
```

When you enter this form of the command, VAX/VMS determines what type of terminal you have and sets several parameters to appropriate values. For example, if you have a VT100 terminal, VAX/VMS sets parameters so that your terminal has lowercase characters and tabs, so that the screen is 80 columns wide by 24 lines high, so that the terminal can buffer characters for the type-ahead feature, and so forth. Since you nearly always need to set the terminal type, you may want to put the SET TERMINAL/INQUIRE command into your LOGIN.COM file; then, the command will be executed every time you log in. (See Section 1.6.11 for information on LOGIN.COM files.)

If you want to determine the current parameter settings for your terminal, use the **SHOW TERMINAL** command. VAX/VMS responds with a list of all of the parameters that can be set for a terminal, and the current settings for your terminal.

At times you may need to inform the system of the speed at which your terminal operates. When doing this, you must tell VAX/VMS the speed to which you will set your terminal *before* you actually set it; if you set the terminal speed first, VAX/VMS will not be able to understand the characters you type. To set the terminal's speed, use the **SET TERMINAL** command with the **/SPEED** qualifier. For example, if you want to change your terminal to run at 4800 bits per second, use the following command:

```
SET TERMINAL/SPEED=4800
```

After you enter this command, VAX/VMS will not recognize any characters that you type until you change your terminal's setting—by means of a manual setting or the **SET-UP** procedure (depending on your terminal type)—to 4800 bits per second.

The **SET TERMINAL** command has many other qualifiers available for more specific purposes. For a complete list of these qualifiers, see the description of the **SET TERMINAL** command in the *Guide to Using DCL and Command Procedures on VAX/VMS*.

1.2.5 Logging In to Other Network Nodes

If you work on a VAX/VMS system with DECnet-VAX software and your system is on a computer network, you can connect your terminal to other computers on the network by entering DCL commands. You must know the name of the computer, or node, that you want to reach, and you must have access to an account there.

You can find out which network nodes are accessible with the **SHOW NETWORK** command. For example:

```
$ SHOW NETWORK
VAX/VMS Network status for local node 2.18 KLEE on 4-JUN-1984 13:51:43.91
```

```
The next hop to the nearest area router is node 2.4 MARC.
```

Node	Links	Cost	Hops	Next Hop to Node
2.18 KLEE	0	0	0	(Local) -> 2.18 KLEE
2.1 BOSS	0	3	1	UNA-0 -> 2.1 BOSS
2.2 ECHO	0	3	1	UNA-0 -> 2.2 ECHO
2.4 MARC	0	3	1	UNA-0 -> 2.4 MARC
...				
2.25 NEST	0	6	2	UNA-0 -> 2.19 FLOSS

```
Total of 12 nodes.
```

For more information on the **SHOW NETWORK** command, see the *Guide to Using DCL and Command Procedures on VAX/VMS*.

To connect your terminal to other network nodes, use the SET HOST command, as in the following example:

```
$ SET HOST MARC
Username: AUGUST
Password:
```

This command connects a terminal to the network node named MARC. After you have successfully connected your terminal to another node, follow the login procedure outlined in Section 1.2.1.

1.3 Using DCL

Once you have logged in to a VAX/VMS system, you can use DCL commands for tasks such as executing programs or printing files. The next two sections present rules and conventions for entering these commands and for defining symbolic names for them.

1.3.1 Rules for Entering and Editing DCL Commands

When entering DCL command lines, you should pay particular attention to the rules in the following list. Other rules for entering DCL command lines are described in the *Guide to Using DCL and Command Procedures on VAX/VMS*.

- You can truncate any command name or qualifier name to four characters. Fewer than four characters are acceptable if the truncated name is unique to the command that you want.
- You must precede each qualifier name with a single slash character (/).
- If you omit a required parameter (for example, a file specification), the VAX/VMS DCL command interpreter will prompt you for it.
- You can enter a command on as many lines as you wish, as long as you end each line (except the last) with a hyphen (-).
- After you have entered a complete command, you must type a RETURN to pass the command to the system for processing.
- You can cancel an entire command, before the final RETURN, by typing <CTRL/Y>.
- You can cancel the current line of a multiline command by typing <CTRL/U>.
- You can interrupt command execution (and user programs) by typing <CTRL/Y>. To resume the interrupted command, enter the CONTINUE command. To stop processing completely after typing <CTRL/Y>, simply do not enter the CONTINUE command. At that point, you can enter other DCL commands or run other programs.

If you enter a command incorrectly (for example, if you misspell a command or qualifier name), the command interpreter issues an error message and you must either retype the entire command or edit the command and then reenter it.

CTRL/B	Recalls successive command lines so that they can be edited.
CTRL/H	Moves the cursor to the beginning of the command line being displayed. (BACKSPACE key generates <CTRL/H>.)
CTRL/E	Moves the cursor to the end of the command line being displayed.
{ CTRL/D ←	Moves the cursor one character to the left.
{ CTRL/F →	Moves the cursor one character to the right.
CTRL/J	Deletes the characters or word to the left of the cursor; that is, characters up to the next blank are deleted. (LINEFEED key generates <CTRL/J>.)
CTRL/A	Alternates between insert and overstrike mode. Depending on the <CTRL/A> setting the characters that you enter either replace existing characters or are added to the command line. For example, after typing <CTRL/B>, you are in overstrike mode by default; to get into insert mode, you type <CTRL/A>; to return to overstrike mode, you type <CTRL/A>; and so on.
CTRL/U	Deletes all characters between the current cursor position and the beginning of the command line.
CTRL/C	Cancels the entire operation.

Command line editing enables you to correct typographical errors and other errors in lengthy command lines and saves you the trouble of reentering the entire line.

1.3.2 Rules for Defining DCL Symbols

One of the most useful features of VAX/VMS is its ability to recognize symbols that represent commands and partial command strings. You define a symbol by placing one of the assignment operators := or ::= between the symbol and the string it represents. If you use the ::= operator, VAX/VMS inserts the symbol in the global symbol table; if you use the := operator, VAX/VMS inserts the symbol in a local symbol table.

The Guide to Using DCL and Command Procedures on VAX/VMS describes symbol tables in detail. Briefly, VAX/VMS creates a global symbol table for you when you log in. Then, each time you execute a command procedure, VAX/VMS creates a new command level and a local symbol table for that level. If one command procedure executes another procedure, VAX/VMS creates another new—lower—command level and another local symbol table, and so forth. Every command level can access the symbols in the global symbol table, as well as symbols in local symbol tables at higher command levels. In other words, you can define local symbols in a command procedure that are available to lower-level procedures. Note that VAX/VMS discards local symbols and their values when a procedure exits. That is, a program or command procedure can create local symbols, but the symbols disappear

when the program or procedure ends. However, you can define local symbols at DCL command level, and these symbols remain until you explicitly remove them or until you log out.

For example, the following command defines the local symbol FORTRAN as a command to invoke the FORTRAN compiler, with two added qualifiers:

```
$ FORTRAN := FORTRAN/G_FLOATING/NOLIST
```

You can pass information to a higher-level command procedure by defining a global symbol to contain the information. Because there is only one global symbol table, and it is accessible at all command levels, the higher-level command procedure can simply test the value of the symbol.

You can tell VAX/VMS that you want to allow abbreviations of a symbol by using the asterisk (*) character to end the acceptable abbreviation. For example, if you wanted to abbreviate the command FORTRAN to FOR, you could use the following command:

```
$ FOR*TRAN ::= FORTRAN/G_FLOATING/LIST
```

Once you have defined this symbol, you can type FOR, FORT, FORTR, FORTRA, or FORTRAN to invoke the FORTRAN compiler with the /G_FLOATING and /LIST qualifiers. Note that the double equal sign in this symbol definition denotes the creation of a global symbol.

You can determine the definition of a symbol by typing SHOW SYMBOL followed by the symbol name. For example:

```
$ SHOW SYMBOL FOR
FOR*TRAN = FORTRAN/G_FLOATING/LIST
```

To delete a symbol, type DELETE/SYMBOL, followed by the symbol name. If you do not specify a symbol table qualifier, LOCAL is assumed. If the symbol is in the global table, type DELETE/SYMBOL/GLOBAL followed by the symbol name. For example, you could delete the symbol FORTRAN, which you had defined previously, by typing:

```
$ DELETE/SYMBOL/GLOBAL FORTRAN
```

Many VAX/VMS users keep a file of symbols for the system to define every time they log in, thus creating a set of personal commands for special purposes. For information on having the system read such a file each time you log in, see Section 1.6.11 on the LOGIN.COM file. Sections 1.6.1 to 1.6.4 contain information on using symbols in command procedures.

1.4 Getting Help from VAX/VMS

You can get help from VAX/VMS on commands, qualifiers, and other keywords by using the HELP command:

```
HELP [topic [subtopic ...]]
```


For instance, to find out about the FORTRAN command, type HELP FORTRAN. If you type only HELP, you get a list of all of the topics for which help is available, and if you type HELP FORTRAN, you get a list of FORTRAN qualifiers, parameters, and other topics for which help is available. You can also have the help text written to a file by specifying the /OUTPUT qualifier and a file specification, after the HELP command. For example, to create a file named CROSS.TXT, containing the HELP information on the /CROSS qualifier to the FORTRAN command, enter:

```
$ HELP/OUTPUT=CROSS.TXT FORTRAN /CROSS
```

See Section 1.5.1 for more information on file specifications.

1.5 Working with Files

In order to name, access, and use your files effectively, you need information about:

- File specifications, which you use to refer to a particular file
- File-handling commands, which you use to perform any of the following operations on files: move, delete, list directory, rename, set protection, search contents, and display or print contents
- Command procedures (including the LOGIN.COM command procedure), which allow you to execute multiple DCL commands both from a terminal and from a batch job

1.5.1 File Specifications

A file specification provides the system with all of the information it needs to uniquely identify a file. The maximum length of a file specification, including all delimiters, is 255 characters. Note that you do not have to enter each field in every file specification; often you can use the system-supplied default values. The fields of a file specification are:

```
node"access-control-string"::device:[directory]filename.type;version
```

For example:

```
VIOLET'ROTHKO MARK'::USERD:[ROTHKO]PAINTINGS.TXT;3
```

The fields are explained below:

node"access-control-string"

When copying files to or from another network node, you must include the node name, followed by two colons.

Depending on your privileges and the operation that you are going to perform, you may have to provide an access control string. An access control string specifies the user name and password to be used on the remote node. If you include the access control string, you must enclose it in quotation marks, and it must precede the two colons. The access control string cannot be longer than 42 characters, and the password cannot be longer than 31 characters.

If the file specification for the remote node does not conform to VAX/VMS syntax, you must enclose everything after the two colons in quotation marks.

device

Your system manager sets up logical names for the storage devices attached to your system. You should use these logical names, rather than physical device names, when referring to files on these devices. For example, the sample file specification shown previously indicates that the logical name USERD has been defined as the name of the device containing the directory [ROTHKO].

directory

A named catalog of files. You specify a directory as a character string of up to 39 alphanumeric characters (dollar sign and underscore are also allowed after the first character) or as a sequence of character strings separated by periods. In each case you must enclose the directory name in square brackets ([]) or angle brackets (<>).

The following examples show how you can refer to directories:

```
[360,015]
[KANDINSKY]
[KANDINSKY.FORT]
```

See Section 1.5.1.3 for a description of directories and directory hierarchies.

filename

A string of up to 39 alphanumeric characters (in addition, optionally, to dollar sign and underscore—with the first character an alphanumeric) that you assign to a file. The following examples show legal file names:

```
FORT_TEST
BUDGETER$
APR23RECS
a
averylongfilename
```

Note that the case of the letters in the file name does not matter—the system recognizes both lowercase and uppercase and does not distinguish between them.

type

A character string that, by convention, describes the contents of the file. For example, the file type .FOR is used to denote FORTRAN source programs. Any alphanumeric characters (in addition to dollar sign and underscore; with first character an alphanumeric), can be used in file types, and the string can be up to 39 characters long. The file type must be preceded in the file specification by a period.

Some of the most common file type conventions are listed in Table 1-1.

version

A decimal integer between 1 and 32767, preceded by a semicolon or a period. When you update or modify a file without specifying the version number of the output file,

the system increments the current version number by 1 when creating the output file. You must specify a version number when you delete a file. You can use a version number of 0, or omit the integer and include only the semicolon or period, to refer to the most recently created version of the file. For example:

```
FORPROG.LIS;5
FORPROG.LIS.5
FORPROG.LIS;0
FORPROG.LIS;
```

Given that FORPROG.LIS;5 is the latest version of the file and thus has the highest version number, all of these file specifications refer to the same file.

Table 1-1 shows the most common file types used by FORTRAN programmers.

Table 1-1: Common File Types

Type	Expected File Contents
FOR	FORTRAN source program; default input type for FORTRAN command
COM	Command procedure file to be executed interactively or as a batch job
DAT	Input or output data file
DIR	Directory file
EXE	Executable image file; default output type for LINK command; default input type for RUN command
HLB	Help text library file
HLP	Input text for help libraries
JOU	Journal file for the EDT editor
LIS	Listing file; default listing output type for FORTRAN command; default input type for the PRINT and TYPE commands
LOG	Batch job output file
MAI	Mail message file
MAP	MAP image map; default listing output type for LINK command
OBJ	Object file; default output type for FORTRAN command; default input type for LINK command
OLB	Object module library
TMP	Temporary file
TXT	Input file for text libraries, mail utility

1.5.1.1 File Specification Defaults

Not all elements of a file specification need to be written each time you use a file. The default values that the system supplies for unspecified elements are summarized in Table 1-2.

For example, if you specify only a file name when compiling a FORTRAN program, the compiler can process the source program if the file meets the following three requirements:

- The file is stored on the default device on the local node.
- It is cataloged under the default directory name.
- It has a file type of FOR.

If more than one file meets these conditions, the compiler processes the file with the highest version number.

For example, assume that your default device is USERD, your default directory is SMITH, and you supply the following file specification CIRCLE to the compiler.

The compiler searches device USERD in directory SMITH, seeking the highest version of CIRCLE.FOR. If you do not specify an output file, the compiler generates the file CIRCLE.OBJ, stores it on device DBA0 in directory SMITH, and assigns it a version number that is 1 higher than any other version of CIRCLE.OBJ currently cataloged in directory SMITH on USERD.

Table 1-2: File Specification Defaults

Element	Default Value
Node	Local network node
Device	User's current default device
Directory	User's current default directory
File type	Depends on usage: Input to compiler FOR Text library input to compiler TLB Output from compiler OBJ Input to linker OBJ Output from linker EXE Input to RUN command EXE Compiler source listing LIS Linker map listing MAP Input to executing program DAT Output from executing program DAT
Version	Input: highest existing version Output: 1, if no existing version, otherwise, highest existing version plus 1

1.5.1.2 Wildcard Characters

You can often substitute characters called *wildcards* for directories, file names, file types, and version numbers in file specifications. There are two types of wildcards: (1) general-purpose wildcard characters used in input file specifications and (2) special wildcard characters used only in alphanumeric directory specification fields. Both types of wildcards can be combined in numerous ways in directory specifications.

The special wildcard characters are ellipsis (...) and minus sign (-). Their use is explained in Section 1.5.1.3.1.

The general-purpose wildcard characters are the asterisk (*) and the percent sign (%).

- An asterisk represents a string of characters of any length from zero to the maximum allowed. For example, *.FOR refers to all files whose file types are FOR. A*.FOR refers to all files with names starting with A or with the name A and with a file type of FOR, such as AXA.FOR.
- A percent sign represents exactly one character. For example, Z%.FOR refers to all files whose types are FOR and whose names have two characters: Z followed by any other valid character.

The following example shows how to use general-purpose wildcards with file specification fields to keep track of groups of files. If you have a number of FORTRAN programs, and you use default file types, you can refer to all of the FORTRAN source files on your current directory with the specification:

```
*,FOR
```

You can refer to the most recent executable versions of programs that exist in subdirectory [FORTRAN.EXECUTE] by specifying a semicolon version number delimiter, but omitting the version number itself:

```
[FORTRAN,EXECUTE]*.EXE;
```

You can refer to help files on the network node HUNTER with the specification:

```
HUNTER::SYS$HELP:*,*
```

Not all commands allow the use of wildcard characters in file specifications. For instance, wildcards cannot be used in the FORTRAN command. Each file that you wish to compile must be explicitly named.

1.5.1.3 Directories and Subdirectories

Directories are “file catalogs,” each of which is owned by a particular user. The system identifies each directory by an alphabetic name, and you use these names in referring to the directory. A directory name consists of up to thirty-nine alphanumeric characters and can contain any characters that are allowed in file names. Directories are just special files that always have a file type of .DIR and a version of 1. Often, the directory that you own has the same name as the account under which you log in.

VAX/VMS allows you to create subdirectories within a directory. Like directories, subdirectories have names. Subdirectory names are written as lists of directory names, separated by periods; the entire list is enclosed in square brackets. The first directory name in

the list represents the highest-level directory; the last name in the list represents the subdirectory to which you are referring, with any intermediate directories listed in between them. For example:

```
$ DIRECTORY [FAULKNER.PROGRAMS.FORT]
```

The top-level directory in this example is FAULKNER, the intermediate subdirectory is PROGRAMS, and the subdirectory is FORT. The DIRECTORY command lists all files in the specified (or default) directory.

VAX/VMS keeps track of your current default device and directory. When you log in, your default device and directory are set to your home disk area. To determine the current default values, use the SHOW DEFAULT command. For example:

```
$ SHOW DEFAULT  
  WORKD:[BRONSTEIN.FORT]
```

You can change these default values with the SET DEFAULT command. For example, if your default directory when you log in is [FAULKNER] and you wish to work on files in the directory [FAULKNER.PROGRAMS], you can avoid typing the directory name with each file name by using the following command:

```
$ SET DEFAULT [PROGRAMS]
```

The files in each directory and subdirectory occupy a certain amount of disk space. If disk quotas are enabled on the disk you work on, you cannot exceed the quota assigned to you. You can use the SHOW QUOTA command to determine how much disk space is occupied by files in a directory or subdirectory. In response to SHOW QUOTA, the system tells you how much disk space you are using, what your quota is, and how much free space is available to you. For example:

```
$ SHOW QUOTA  
  User [BRONSTEIN] has 278 blocks used, 722 available, of 1000  
  authorized and permitted overdraft of 50 blocks on WORKD
```

If no disk quotas are in effect for your disk, the SHOW QUOTA command informs you that the disk quota accounting file is not active.

Creating a Subdirectory

To create a subdirectory, use the CREATE/DIRECTORY command. For example, if you wish to create the subdirectory [FAULKNER.PROGRAMS.FORT], enter the following command:

```
$ CREATE/DIRECTORY [FAULKNER.PROGRAMS.FORT]
```

If you have already executed the command SET DEFAULT [FAULKNER.PROGRAMS], your current default directory is [FAULKNER.PROGRAMS]. This would enable you to use the following, shorter, form of the command:

```
$ CREATE/DIRECTORY [FORT]
```

After you execute this command, the directory [FAULKNER.PROGRAMS] will contain the file FORT.DIR, which is the catalog of the files in [FAULKNER.PROGRAMS.FORT].

You can use the VAX/VMS facility for creating directories and subdirectories to construct trees—hierarchies—of logically related files. The number of directories you can create is limited only by your available disk space; however, no hierarchy can contain more than eight levels. Figure 1-2 illustrates the concept of directory hierarchies.

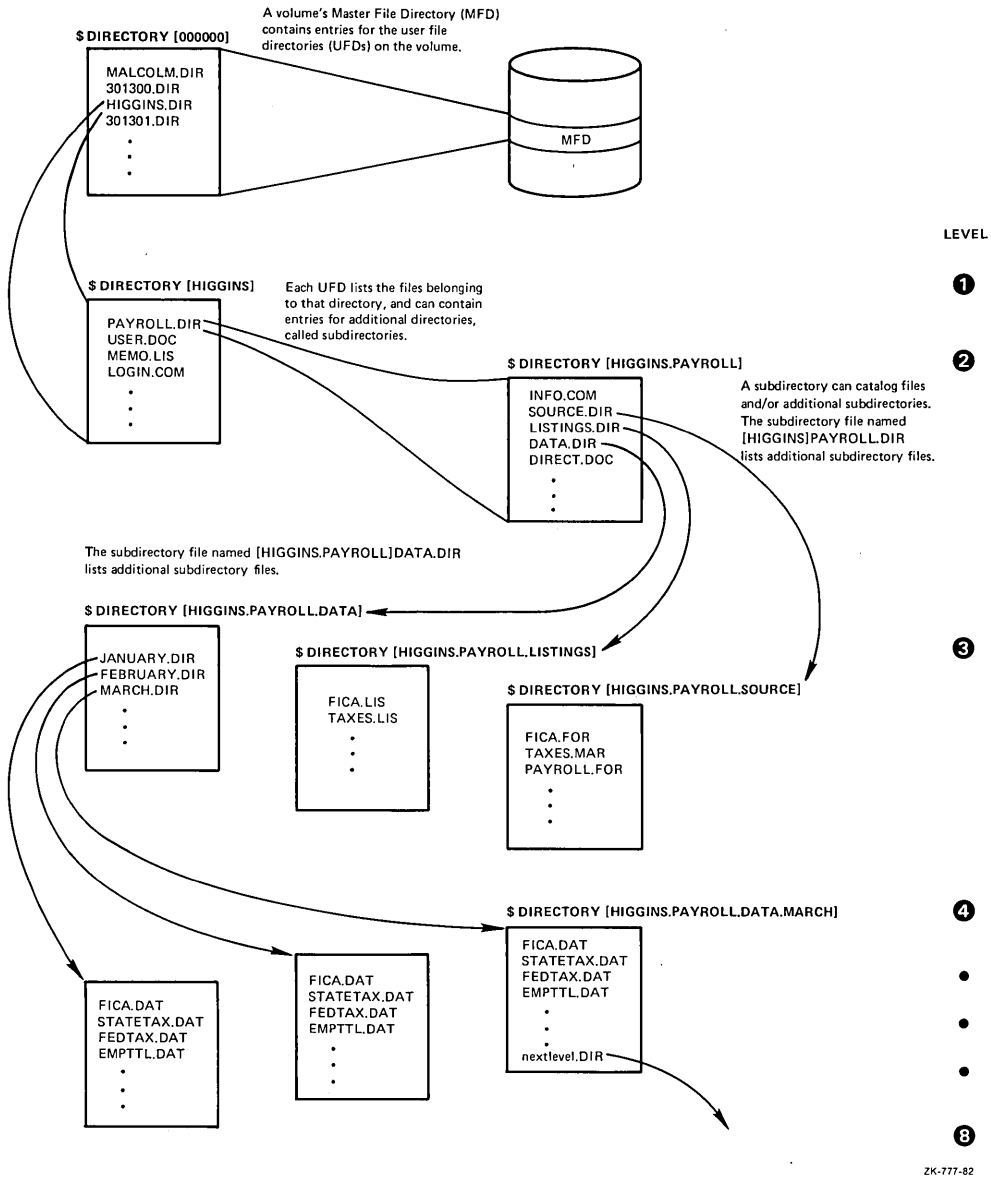


Figure 1-2: A Directory Hierarchy

You may often refer to files that are somewhere in your directory hierarchy but not in your current default directory. Two special symbols exist to make references easier: the ellipsis (...) and the hyphen or minus sign (-). The ellipsis is used to search down a directory hierarchy, and the hyphen is used to search up the hierarchy.

For example, [BRONSTEIN...] refers to the directory [BRONSTEIN] and all of the subdirectories in the hierarchy below [BRONSTEIN]. The directory specification [...FORT] refers to all subdirectories named FORT below the current default directory. Since you can specify the current default directory with [], you can refer to the entire hierarchy under your current default directory with [...].

Hyphens allow you to search up the hierarchy one directory at a time; each hyphen stands for one hierarchical level. If, for example, your current default directory is [BRONSTEIN.FORT.TEST], you can refer to [BRONSTEIN] by [--]. The directory specification [-] refers to [BRONSTEIN.FORT], and [-.SOURCES] refers to [BRONSTEIN.FORT.SOURCES].

Deleting a Subdirectory

To delete a subdirectory properly, you must first remove any files in it. Then, you can delete the DIR file. Note that directory and subdirectory files are protected to prevent accidental deletion; you cannot delete them without first resetting the file protection. (For more information on file protection, see Section 1.5.2.5.)

1.5.1.4 Logical Names

Logical names are a shorthand way of specifying device, directory, or file names to which you refer frequently.

Logical names are alphanumeric character strings, up to 255 characters long.

Every logical name is paired with one or more equivalence name. An equivalence name can be a file specification, part of a file specification (a device or a device and a directory), or another logical name. When programs and command procedures refer to physical files by logical name, VAX/VMS translates the logical name into its equivalence name.

Logical names that translate to more than one equivalence name are called search lists. Search lists are useful in referring to groups of files that are not collected together in a single directory.

The remainder of this section contains information about the following topics:

1. Defining logical names
2. Logical name tables
3. Predefined logical names
4. Deleting and showing logical names
5. Assigning a logical name with the MOUNT command
6. Search lists

The topics are addressed, in the order shown here, under the headings that follow.

Defining Logical Names

Two VAX/VMS commands, ASSIGN and DEFINE, equate a logical name and an equivalence name. The commands differ in two respects: (1) the order in which they require the logical name and the equivalence name and (2) if you supply a terminating colon with the logical name, ASSIGN removes it, whereas DEFINE does not.

The following example shows how to equate a logical name, TEST, and a file specification, using the ASSIGN and DEFINE commands:

```
# ASSIGN USERD:[JUNG.FORT]FORTEST.FOR TEST
# DEFINE TEST USERD:[JUNG.FORT]FORTEST.FOR
```

The preceding commands equate the logical name TEST to the file specification USERD:[JUNG.FORT]FORTEST.FOR. After you have issued one of these commands, you can then refer to the file by the name TEST. For example:

```
# TYPE TEST
```

You can also run programs and execute command procedures that refer to the logical name TEST. For example, suppose that you have written a FORTRAN program that opens a file using the following code:

```
OPEN (UNIT=1, FILE='TEST', ORGANIZATION='SEQUENTIAL', STATUS='OLD')
```

When you run your program and this line is executed, FORTRAN passes the string TEST to VAX/VMS as the file specification. In searching for the corresponding file, VAX/VMS first tries to translate the string TEST. If TEST is a logical name, the translation yields an equivalence name, which is translated in turn. When no further translations are possible, VAX/VMS assumes that it has a file specification, and your program opens the file USERD:[JUNG.FORT]FORTEST.FOR. You can specify a different file when you run the program again by issuing another ASSIGN (or DEFINE) command to create a new equivalence name for the logical name TEST. For example:

```
# ASSIGN WORKD:[JUN]REAL.DAT;7 TEST
```

If you enter the preceding ASSIGN command and then rerun your program, the program opens the file WORKD:[JUN]REAL.DAT;7.

Logical Name Tables

Logical names are kept in logical name tables. Five specific logical name tables—USER, PROCESS, JOB, GROUP, and SYSTEM—are of special interest.

- User logical name tables contain logical names that are local to the program that is currently executing. They automatically go away when the execution of the program ends. Use the ASSIGN/USER command to put a logical name into the user table.
- Process logical name tables contain logical names that are local to your process; the ASSIGN and DEFINE commands place logical names in the process logical name table by default.

- Job logical name tables contain logical names that are local to your current process and any descendent processes created using the SPAWN command. You must use the /JOB qualifier to refer to the job logical name table.
- Group logical name tables can be used by anyone in a user group (as defined by the group field in the user identification code); you must use the /GROUP qualifier to refer to the group logical name table.
- The system logical name table contains entries that can be accessed by any process in the system; you must use the /SYSTEM qualifier to refer to the system logical name table.

To place logical names in the group table, you need the GRPNAM privilege; to place logical names in the system table, you need the SYSNAM privilege.

See the *VAX/VMS DCL Dictionary* for detailed information on the use of logical name tables.

Predefined Logical Names

The operating system supplies a number of predefined logical names that are already associated with particular file specifications. Table 1-3 lists the logical names of special interest to FORTRAN users.

Table 1-3: Predefined System Logical Names

Name	Meaning	Default
SYS\$COMMAND	Default command input stream	User's terminal (interactive); batch command file (batch)
SYS\$DISK	Default device	As specified by the user
SYS\$ERROR	Default error message output file	User's terminal (interactive); batch log file (batch)
SYS\$INPUT	Default input stream	User's terminal (interactive); batch command file (batch)
SYS\$NODE	Current local node	Network node name for the local system if DECnet-VAX is active on the system
SYS\$OUTPUT	Default output stream	User's terminal (interactive); batch log file (batch)
SYS\$LOGIN	User's output file for system	Established by system manager
SYS\$SCRATCH	Default device and directory for scratch files created by the compiler	Established by the system manager

Deleting and Showing Logical Names

To delete a logical name from a logical name table, use the DEASSIGN command. For example:

```
$ DEASSIGN TEST
```

The preceding command deletes the logical name TEST from the process logical name table. To remove logical names from the job, group, and system logical name tables, you must use the /JOB, /GROUP, and /SYSTEM qualifiers. You need the GRPNAM privilege to remove a logical name from the group table and the SYSNAM privilege to remove a logical name from the system table. (Note: All logical names are deleted from the user logical name table each time a program completes. See Section 1.5.1.3.)

You can ask VAX/VMS to show you the equivalence name for a logical name with the SHOW LOGICAL command. This command accepts the logical name as a parameter. If repeated translation is necessary (that is, if the logical name stands for another logical name, and so on), up to 10 translations are performed.

The SHOW TRANSLATION command, on the other hand, does not do repeated translation; translation is repeated until an equivalence name is found that is not, itself, a logical name. The response to SHOW TRANSLATION is the equivalence name that corresponds to the first occurrence of that logical name in the logical name tables. The tables are searched in the following order: process, job, group, and system. For example, suppose you define TEST with the following ASSIGN command:

```
$ ASSIGN TEST_FILE TEST
```

If TEST__FILE is a logical name pointing to USERD:[JUNG.FORT]FORTEST.FOR, the responses to the SHOW LOGICAL and SHOW TRANSLATION commands are as follows:

```
$ SHOW TRANSLATION TEST
TEST = "TEST_FILE" (LNM#PROCESS_TABLE)
$ SHOW LOGICAL TEST
TEST = "TEST_FILE" (LNM#PROCESS_TABLE)
TEST_FILE = "USERD:[JUNG.FORT]FORTEST.FOR" (LNM#PROCESS_TABLE)
```

Assigning a Logical Name with the MOUNT Command

You can specify a logical name as a parameter of the MOUNT command when mounting a volume on a device. The MOUNT command has the form:

```
$ MOUNT device-name,... [volume-label,...] [logical-name[:]]
```

If your program refers to devices by means of logical names, you can change the association between the device name and the logical name when you mount the device. For example:

```
$ MOUNT MTA0: TAPE2 MYTAPE
```

The preceding command associates the logical name MYTAPE with device name MTA0 and volume label TAPE2. Whenever your program refers to logical name MYTAPE, access is to the volume labeled TAPE2 mounted on a system-assigned magnetic tape unit. If you

subsequently mount a different tape to be referenced by the logical name MYTAPE, you can change the logical name association when you issue the MOUNT command. For example:

```
$ MOUNT MTA0: TAPE7 MYTAPE
```

The preceding command associates the logical name MYTAPE with device name MTA0 and volume label TAPE7.

Search Lists

Search lists are logical names that translate to an ordered list of equivalence names, rather than to a single name. You create search lists by specifying a list of equivalence names, separated by commas, in place of the single equivalence name in the ASSIGN and DEFINE commands. The translation behavior of search lists depends on whether they are being used for input files or output files. The typical user of search lists is to collect, under one logical name, groups of files that are in different places. For example, suppose you are working on two projects, called A and B. Suppose that the important files for project A are kept in directory [A.LIB] and those for project B, in directory [B.LIB]. If you wish the system to search for files first in [ROTHKO] and then in [A.LIB] and then in [B.LIB], you can define a search list with the following command:

```
$ DEFINE SRC [ROTHKO],[A.LIB],[B.LIB]
```

The preceding command establishes the logical name SRC as a search list with the equivalence names of [ROTHKO], [A.LIB], and [B.LIB], in that order. You can then use the name SRC whenever you want the system to search these directories for a specified file. For example, the command

```
$ FORTRAN SRC:FOURIER
```

searches for the file FOURIER.FOR in [ROTHKO] first; then, if no file by that name is found there, searches [A.LIB]; and finally, if no such file is found there, searches [B.LIB]. The search terminates in the first place that FOURIER.FOR is found, and that file is used for the input to the FORTRAN command.

If a search list is used for an output file specification, the new file is created in the directory specified by the first name in the list. For example, if FOURIER.FOR existed only in [A.LIB], the following command would compile [A.LIB]FOURIER.FOR and create the object file [ROTHKO]FOURIER.OBJ:

```
$ FORTRAN/OBJECT=SRC: SRC:FOURIER
```

1.5.2 File-Handling Commands

The most common file-handling operations are:

- Moving files
- Deleting files
- Listing and changing file names

- Handling file protections
- Displaying the contents of files on your terminal or printing them on a line printer

The following sections describe the commands used to perform these operations. Note that these commands affect entire files; they do not act on the contents of the files. (See the *Guide to Using DCL and Command Procedures on VAX/VMS* for complete descriptions of all DCL commands.)

1.5.2.1 Moving Files

Two DCL commands, COPY and APPEND, move files:

- COPY makes a new copy of the file in a directory you specify, thus allowing you to move the file to a new device if you wish.
- APPEND copies the contents of the files that you specify into a single file.

In both cases, the original files are retained; they must be explicitly deleted if you do not want to keep them.

(Note: The RENAME command does not move files; it simply changes the directory entry for the file. For a description of RENAME, see Section 1.5.2.4.)

COPY Command

To copy a file from one directory to another or from one device to another, use the COPY command, listing the input file specification(s) first and the output specification second. Only one output specification is allowed. However, you can use wildcards in the output specification to copy a group of files to another device or directory. For example:

```
$ COPY WORKD:[BACKUS.FORT]A*.*,B*.* [MINE]*.*
```

The preceding command copies every file in WORKD:[BACKUS.FORT] that has a name starting with A or B to the directory WORKD:[MINE]. The files retain their names.

If you are on a DECnet network, you can also use the COPY command to copy files from one network node to another. For example:

```
$ COPY RIDER::USERD:FILE.TXT *.*
```

The preceding command copies FILE.TXT from device USERD on node RIDER to the current default directory.

APPEND Command

To copy two or more files into a single file, use the APPEND command. APPEND concatenates the files you specify as input, in the order in which you have listed them. You can add the input files to an existing file or to a new file. For example:

```
$ APPEND PARSER.FOR,HELPER.FOR,MYPROG.FOR TEST.FOR/NEW
```

The preceding command creates a new file called TEST.FOR consisting of the three FORTRAN source files in the order shown. The /NEW qualifier is required if you want to create a new file.

You can also use wildcard characters to concatenate files with the APPEND command. For example:

```
$ APPEND *.TXT WHOLE.TXT
```

When you execute the preceding command, all files of type TXT are appended to the existing file WHOLE.TXT. If WHOLE.TXT did not exist, you would get an error message, unless you used the /NEW qualifier.

1.5.2.2 Deleting Files

Two DCL commands, DELETE and PURGE, delete files.

DELETE Command

The DELETE command eliminates the specified files from the disk and makes the space they occupied available for other files. You cannot retrieve files once you have deleted them; therefore, you should be as explicit as possible when specifying files to be deleted. In order to minimize the number of deletions performed by mistake, VAX/VMS requires information about the version numbers of the files to be deleted.

You can delete the latest version in the specified directory by using a version number of 0 or by supplying only the semicolon. For example, the following command deletes the most recent version of TEST.FOR, leaving earlier versions intact.

```
$ DELETE TEST.FOR;
```

To delete all versions of TEST.FOR, use the following command:

```
$ DELETE TEST.FOR;*
```

PURGE Command

The PURGE command deletes old versions of a file. It allows you to specify how many of the most recent, or highest numbered, versions you want to keep. By default, it deletes all versions except the most recent.

PURGE does not require you to enter version numbers. For example, the following PURGE command deletes all but the latest version of TEST.FOR:

```
$ PURGE TEST.FOR
```

The following PURGE command is more general. It purges all but the latest version of every file in every directory and subdirectory in the hierarchy under the directory [JONES].

```
$ PURGE [JONES...]
```

To keep more than the latest version of files in directories you purge, you must use the /KEEP qualifier to indicate the number of versions that you wish to retain. For example, the following command retains the three latest versions of every file in the directory [JONES], and all subdirectories below it as well (if that many versions exist).

```
$ PURGE/KEEP=3 [JONES...]
```

If there are fewer than three versions of a file, none of the existing versions are deleted.

To allow extra checking when deleting files, use the /CONFIRM qualifier. To verify what files are being moved or deleted, use the /LOG qualifier.

1.5.2.3 Listing File Names

The DIRECTORY command lists the files in the specified directory in alphabetical order. If you omit a directory specification, typing the DIRECTORY command lists the files in the current default directory. To get additional information, or to format the output, you can add qualifiers to the DIRECTORY command. For example:

```
$ DIRECTORY/COLUMNS=1/DATE=CREATED/PROTECTION
```

The preceding command tells VAX/VMS to list the files in your directory in a single column instead of in the default four columns, and to include the date of creation and the protection in effect for each file.

You can obtain all of the information available on a file or group of files with the /FULL qualifier of the DIRECTORY command. This qualifier causes the following attributes to be listed for each file:

- File name
- File type
- Version number
- Number of blocks used
- Number of blocks allocated
- Date of creation
- Date of last backup
- Date last modified
- Date of expiration
- File owner's UIC
- File protection
- File identification number (FID)
- File organization
- Other file attributes
- Record attributes
- Record format
- Access control list (ACL)

These items are described in the *VAX Record Management Services Reference Manual*.

You can use the /TOTAL qualifier to find out how many files exist in a directory. You can use the /SIZE qualifier to find out how many blocks of disk space each file takes up. If you combine these qualifiers, you can find the total number of files and the total space used, without getting a list of the file names. For example:

```
$ DIR/SIZE/TOTAL
```

```
Directory USERD:[THOMPSON.DOCTOR]
```

```
Total of 191 files, 5169/5409 blocks.
```

The *Guide to Using DCL and Command Procedures on VAX/VMS* contains a complete list of the qualifiers you can use with DIRECTORY. You can also type HELP DIRECTORY to get a list of qualifiers.

1.5.2.4 Renaming Files

You can use the RENAME command to change the directory specification, file name, file type, and file version of an existing file. For example, the following command changes the directory and file name of the file AVERAGE.OBJ:

```
# RENAME AVERAGE.OBJ [FAULKNER]OLDAVERAGE.OBJ
```

The following RENAME command changes the directory specification of the latest version of the file SAVE.DAT from [FORTRAN.SOURCES.TEST] to [FORTRAN.SOURCES], one level higher in the directory hierarchy:

```
# RENAME [FORTRAN.SOURCES.TEST]SAVE.DAT [-]
```

1.5.2.5 Handling File Protections

File protections allow you to regulate access to your files. Access is granted according to the class of the user who requests it. Users can be in one of four classes: SYSTEM, OWNER, GROUP, and WORLD. SYSTEM users have low group numbers (the exact range is established by the system manager), and are usually system programmers, system managers, and operators. OWNER is the user with the same user identification code (UIC) as the creator of the file. GROUP users are those who have the same group number (the first number in the UIC) as the file's creator. WORLD users are those who do not belong to any of the other categories.

You can use the SET PROTECTION command to establish protection levels for each class of users. Class names can be spelled out or abbreviated to a single letter. Protection codes are R (read), W (write), E (execute), and D (delete). For example, you might want to give system users and your project members (who are in your UIC group) complete access to your file A.FOR, and give others only read access. The following command accomplishes this:

```
# SET PROTECTION=(S:RWED,O:RWED,G:RWED,W:R) A.FOR
```

You can prevent files from being deleted by setting their protections so that deletion is not allowed. For example:

```
# SET PROTECTION A.FOR/PROTECTION=(S:RWE,O:RWE,G:RE,W),-  
#_ B.FOR/PROTECTION=(O:RWE,G:RE,W:R)
```

The preceding command sets protection on two files, A.FOR and B.FOR. A.FOR is protected so that system users and the owner have read, write, and execute access to the file; group members can read and execute the file, but world users have no access to it. B.FOR is protected so that the owner has read, write, and execute access, group members have read and execute access, and world users have read access to it. Because the class SYSTEM was omitted from this command, system users have the same access that they had before the command was entered. Note that this command was continued on a second line with the use of the hyphen continuation character.

If you have access to a file, you can determine its current protection by using the DIRECTORY command with the /PROTECTION qualifier. See Sections 1.5.1.3 and 1.5.2.3 for information about the DIRECTORY command.

The SET PROTECTION/DEFAULT command allows you to choose the protection codes to be assigned to files you create in the current terminal session or batch job. For example,

```
$ SET PROTECTION=(G,W) /DEFAULT
```

The preceding command tells VAX/VMS to give no access to group or world users for any files you create during the current session.

1.5.2.6 Searching File Contents

The SEARCH command allows you to search the contents of a file or a collection of files for a specified text string or strings. You can control how SEARCH finds matching strings and what it tells you about them, and you can look at the lines surrounding the line where the match occurred. For example, you can find all occurrences of a certain variable name in any file whose type is FOR, or you can look at a particular line from a compilation listing file. Type HELP SEARCH or see the *Guide to Using DCL and Command Procedures on VAX/VMS* for descriptions of all of the qualifiers you can use with SEARCH.

Suppose you need to know which routines in all FORTRAN source files in your current default directory contain references to the variable Too_Many_Files. The following example shows how you can search the specified files:

```
$ SEARCH *.FOR Too_Many_Files

* ***** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** *
USERD:[HST,FORT]INITVARS.FOR;2

Too_Many_Files = True

*** ***** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** *
USERD:[HST,FORT]OPENFILE.FOR;1

      IF (Too_Many_Files .EQ. True) THEN
```

If you wanted to see the two lines following the reference to Too_Many_Files in OPENFILE.FOR, you could use the /WINDOW qualifier as follows:

```
$ SEARCH/WINDOW=(0,2) OPENFILE.FOR "Too_Many_Files"
      IF (Too_Many_Files .EQ. True) THEN
          PRINT *,'Too many files,'
      END IF
```

1.5.2.7 Printing and Typing Files

The PRINT and TYPE commands allow you to look at the contents of a file. If you want to look at the file contents on a line-printer listing, use the PRINT command. If you want to look at the contents on your terminal screen, use TYPE. For example:

```
$ PRINT *,LIS/COPIES=10
```

The preceding command causes VAX/VMS to group all files whose file type is LIS into a single print job, and then queue the job to be printed. When the printer is on line and free, 10 copies of each file will be printed.

The /AFTER qualifier of the PRINT command allows you to queue files for printing at a later time. To specify when you want the files printed, follow the /AFTER qualifier with the time (in 24-hour format) when you want the file printing to begin. For example:

```
$ PRINT TEST.LIS/AFTER=20:10
```

The preceding command queues the file TEST.LIS for printing at 8:10 p.m. At the time specified, the job is released from the queue for printing. For more information on how to specify times, see the *Guide to Using DCL and Command Procedures on VAX/VMS* or type HELP SPECIFY DATE__TIME.

The TYPE command causes VAX/VMS to display the file on your terminal. You can control the output with the commands <CTRL/S>, <CTRL/Q>, and <CTRL/O>. When you type <CTRL/S>, VAX/VMS stops sending characters to your terminal; when you type <CTRL/Q>, VAX/VMS starts sending characters again. Thus, you can scan a file quickly until you reach something of interest, type <CTRL/S> and examine the current screen carefully, and then type <CTRL/Q> to begin scanning again. (If your terminal has a NO SCROLL or HOLD SCREEN key and the key is enabled, you can use it like <CTRL/S> and <CTRL/Q> to stop and start the display of your file.)

To suppress the display of output without halting execution of the command, type <CTRL/O>. <CTRL/O> causes the output from the executing command to be discarded instead of being displayed. If you type <CTRL/O> again before the command finishes executing, the display of output is resumed.

When you use TYPE with a group of files, you can use <CTRL/O> to examine only the beginning of each file. For example, suppose you enter the following command:

```
$ TYPE [-]*.FOR
```

The preceding command searches for files with the file type FOR in the directory that is one hierarchy level up from the current default directory. Any files found are displayed on the terminal. If you type <CTRL/O> while the first one is being displayed, the rest of the file is suppressed and the display of the second is begun. Thus, you can look at the beginning of the next file by typing <CTRL/O> after the first few lines of the current file.

1.6 Using Command Procedures

VAX/VMS enables you to group DCL commands into files and execute them as a unit. These files are called command procedures and they have the default file type COM. You can create a command procedure with a text editor, a program, or another command procedure. Once a command procedure exists, you can execute it by appending its file name to the @ (Execute) or SUBMIT commands or the STATUS='SUBMIT' qualifier on the FORTRAN OPEN or CLOSE statement.

You should use a command procedure whenever you want to execute a group of commands repeatedly. In its simplest form, a command procedure is just a file containing a DCL command on each line, preceded by a “\$” character. Comments can be interspersed with the commands by preceding them with exclamation point (!) characters. If you wish to execute a command procedure interactively, use the @ command, for instance:

```
$ @COMFILE
```

The preceding command begins execution of the command procedure contained in the file COMFILE.COM. If you wish to execute a command procedure as a separate, batch process, use the SUBMIT command or specify the command file as the STATUS qualifier of a FORTRAN OPEN or CLOSE statement. For example:

```
$ SUBMIT COMFILE.COM
```

The preceding command submits the file COMFILE.COM as a batch job. See the *Guide to Using DCL and Command Procedures on VAX/VMS* or type HELP SUBMIT for more information about the SUBMIT command.

When you use the @ command, the system by default does not display the commands as they are executed. To have the system display the commands as they are executed, enter the SET VERIFY command before executing the command file. Enter a SET NOVERIFY command to return the system to its default, or silent, behavior.

1.6.1 Using Symbols

Symbols in command procedures are alphanumeric character strings that represent numeric, character-string, or logical values. For example, you can define a symbol to represent a DCL command with some particular set of options you use frequently:

```
$ MD := DIRECTORY/COLUMNS=1/SIZE/PROTECTION
```

Once you have created the symbol MD with this command, typing MD in response to the VAX/VMS prompt is the same as typing:

```
$ DIRECTORY/COLUMNS=1/SIZE/PROTECTION
```

You can create a symbol and assign a value to it in a command procedure by:

- Equating a symbol name to a constant value or another symbol name using an assignment operator (:= or ==) (See Sections 1.6.2 and 1.6.3.)
- Passing parameters and fixed data to a command procedure or a batch job (See Sections 1.6.5 and 1.6.6.)
- Using the INQUIRE and READ commands to prompt for a value for the symbol (See Section 1.6.7.)

A symbol name can be up to 255 characters long, and can contain letters, underscores (—), and dollar signs (\$). Lowercase letters are translated to uppercase by the command interpreter.

1.6.2 Assigning Character Values to Symbols

To assign a character-string value to a symbol, use one of the following forms of the assignment command:

```
symbol-name := character-string-value symbol-name ::= character-string-value
```

Note that one equal sign defines a local symbol; two equal signs define a global symbol. Section 1.3.2 contains more information on defining both kinds of symbols.

The character-string value can contain any alphanumeric or special characters, but if it contains leading spaces or tabs, multiple spaces or tabs, lowercase letters, or any other characters not legal in a symbol name, you must enclose it in double quotation marks. If you need to specify a string that contains quotation marks, you must enclose the entire string in quotation marks and include a double set at the point where you want literal quotation marks to appear. For example:

```
HELLO := "JOHN SAYS "HI" "
```

This command assigns the value JOHN SAYS "HI" to the symbol HELLO.

You can indicate a null string by using a double set of quotation marks with nothing between them, or by specifying no string at all. Both of the following examples specify the null string:

```
NULLSTRING := ""  
NULLSTRING :=
```

1.6.3 Assigning Numeric Values to Symbols

To equate a symbol name to a numeric value, use one of the following forms of the assignment command:

```
symbol-name = expression  
symbol-name ::= expression
```

An expression can be any literal numeric value or an arithmetic or logical expression. As with character values, one equal sign defines a local symbol; two equal signs define a global symbol.

Table 1-4 lists the operators you can use in forming expressions with symbols.

Table 1-4: Summary of Operators in Expressions

Operator		Precedence	Operation
Logical Operators	.OR.	1	Logical OR
	.AND.	2	Logical AND
	.NOT.	3	Logical complement
Arithmetic Comparison Operators	.EQ.	4	Arithmetic equal to
	.GE.	4	Arithmetic greater than or equal to
	.GT.	4	Arithmetic greater than
	.LE.	4	Arithmetic less than or equal to
	.LT.	4	Arithmetic less than
	.NE.	4	Arithmetic not equal to
String Comparison Operators	.EQS.	4	String equal to
	.GES.	4	String greater than or equal to
	.GTS.	4	String greater than
	.LES.	4	String less than or equal to
	.LTS.	4	String less than
	.NES.	4	String not equal to
Arithmetic Operators	+	5	Arithmetic sum
	-	5	Arithmetic difference
	+	7	Arithmetic unary plus
	-	7	Arithmetic unary negate
	*	6	Arithmetic product
	/	6	Arithmetic quotient
String Operators	+	5	String concatenation
	-	5	String reduction

The following examples demonstrate how to form expressions with the operators listed in Table 1-4.

1. `3 .OR. 5`

This expression has the numeric value 7, which is the result of a logical OR operation on the values 3 and 5.

2. `A .EQS. B`

This expression compares the values of string symbols A and B. It has the logical value FALSE. Such logical values, whether TRUE or FALSE, can be assigned to another symbol or used as the condition for an IF...THEN statement. See Section 1.6.4 for information on the IF...THEN statement.

3. 1 .GT. 2

This expression compares the numeric values 1 and 2, and has the value FALSE.

4. "MAYBE" .LTS. "maybe"

This expression compares two strings. String comparisons start with the leftmost character and compare the ASCII hexadecimal values. Since uppercase letters have lower ASCII values than lowercase letters, "MAYBE" is less than "maybe", and the expression has the value TRUE.

1.6.4 Symbol Substitution

The command interpreter substitutes the current values of symbols for the symbol names in a command string. In some contexts, you need to tell the command interpreter to perform symbol substitution; in other contexts, substitution is automatic. For a full description of the command interpreter's symbol substitution algorithm and examples of the results, see the *Guide to Using DCL and Command Procedures on VAX/VMS*. The basic substitution rules are as follows:

- Automatic substitution occurs in IF...THEN and WRITE statements and on the right side of arithmetic assignment statements. In these contexts, the command interpreter assumes that strings starting with an alphabetic character are symbols, and strings starting with a number are numeric literals.

For example, in the following IF...THEN statement, A and B are assumed to be symbols, and the command interpreter substitutes their values when evaluating the expression:

```
IF A .EQ. B THEN GOTO NEXT
```

The arithmetic assignment statement that follows increments the value of the numeric symbol COUNT. The command interpreter substitutes the value of COUNT on the right side only.

```
COUNT = COUNT + 1
```

- Automatic substitution occurs for command synonyms. If the first word on a command line is recognized as a symbol, its value is automatically substituted in the command line.

For example, suppose you have entered the following definition for the symbol ME:

```
$ ME ::= SET DEFAULT SYS$LOGIN
```

Once the symbol ME is defined, you can treat ME as if it were a DCL command. Since the command interpreter automatically substitutes SET DEFAULT SYS\$LOGIN for ME, the effect of typing ME is to set your default directory to your login directory

Suppose you have entered the following definition:

```
$ PDEL ::= DELETE SYS$PRINT/ENTRY=
```

Then you could use PDEL to delete entry number 181 from the SYS\$PRINT queue:

```
$ PDEL 181
```

Note that the space following PDEL delimits the symbol; the command interpreter would not recognize PDEL181. Delimiters can be any of the characters that are not legal symbol-name characters, for example, parentheses: `$ PDEL(181,182,183)`

Since the left parenthesis is not a legal symbol-name character, it is interpreted as a delimiter. Therefore, this command causes entries 181, 182, and 183 to be deleted from SYS\$PRINT.

- Symbols on the right side of nonarithmetic assignment statements and symbols used in place of command parameters or qualifiers must be enclosed by apostrophes (') to force the command interpreter to substitute current values for the symbol.

In the following example, the apostrophes force the command interpreter to substitute the value of the symbol FILENAME for the symbol itself before performing the assignment. If the apostrophes were omitted, OLDSTRING would be assigned the value FILENAME instead.

```
OLDSTRING := 'FILENAME'
```

- Symbols inside quoted strings must be preceded by two apostrophes and followed with a single apostrophe to force the command interpreter to substitute the symbol values for the symbols.

For example, the following assignment statement gives PRINT__STRING a value based on the current value of the symbol FILENAME:

```
PRINT_STRING := "Creating file 'FILENAME'.TST"
```

1.6.5 Passing Parameters to Command Procedures

You can pass up to eight parameters to a command procedure by including them on the command line following the @ (Execute) command and the command procedure name. When the command interpreter executes the command line, it assigns the parameter values supplied to symbols P1 through P8. If you specify only one parameter, it is assigned to P1; P2 through P8 receive null values. If you specify two values, they are associated with P1 and P2, and so on. You can use a pair of double quote characters to specify a null value for a particular parameter.

The following example shows how to pass a value for the parameter P1 to a command procedure. Suppose you want to use a command procedure named FORTEST in your default disk area to compile, link, and run a program that you call NEWTEST.FOR. You would enter the following DCL command:

```
$ @FORTEST NEWTEST
```

Upon receiving this string, the command interpreter assigns the value NEWTEST to the symbol P1. FORTEST can then request the value of P1 when it needs the file name.

The following example shows how FORTEST might refer to the file name that you entered. This DCL command appends the string FOR to the file name and compiles the file named by the resulting string.

```
$ FORTRAN/WARNING/LIST 'P1'.FOR
```

The single quotes around P1 tell the command interpreter to perform symbol substitution. Without the single quotation marks, the command procedure would attempt to compile the file P1.FOR.

When you pass a number of parameters, you must separate them with one or more spaces, as in the following example:

```
$ @RUNPROGS PROGA PROGB PROGC PROGD
```

When it receives this command line, the command interpreter assigns the value PROGA to P1, PROGB to P2, PROGC to P3, and PROGD to P4.

You can pass parameters to a batch job with the PARAMETERS qualifier of the SUBMIT command. If you have more than one parameter to pass, enclose the list in parentheses and separate the parameters with commas. For example:

```
$ SUBMIT RUNPROGS /PARAMETERS=(PROGA,PROGB,PROGC,PROGD)
```

1.6.6 Passing Fixed Data to Programs

You can pass fixed data to a program by including the data in a command procedure that runs the program. (See Section 1.6.7 for a description of how to pass data that is not fixed.)

The following command procedure illustrates how to include fixed data in a command procedure. Note that the data begins on the first line after the RUN command, and that the lines do not begin with a dollar sign. The first line that begins with a dollar sign signals the end of the data.

```
$ FORTRAN AVERAGE
$ LINK AVERAGE
$ RUN AVERAGE
33
66
99
9999
$ DELETE AVERAGE.OBJ;
```

If a line of data to be passed to a program must begin with a dollar sign, you must use the DCL commands DECK and EOD to begin and end the data. For example:

```
$ FORTRAN MESSAGE
$ LINK MESSAGE
$ RUN MESSAGE
$ DECK
This is a line of data that doesn't begin with a $.
$ 30.02
$ EOD
$ DELETE MESSAGE.OBJ;
```

In the preceding example, both lines between the DECK and EOD commands are passed to the program as data.

1.6.7 Controlling Command Procedure Input/Output

You control command procedure input and output (I/O) operations by assigning appropriate values to the system logical names `SYS$COMMAND`, `SYS$INPUT`, and `SYS$OUTPUT`. The following list describes the rules VAX/VMS uses to assign these logical names, and explains how you can reassign them to perform special tasks:

- `SYS$COMMAND` is the logical name for the source of input to the command interpreter. `SYS$COMMAND` does not change: if you execute a procedure interactively, `SYS$COMMAND` is assigned to your terminal; if a batch job executes the command procedure, `SYS$COMMAND` is assigned to the batch file.
- `SYS$INPUT` is the logical name for the default input file. When you execute a command procedure, `SYS$INPUT` is assigned to the command procedure file. When programs request data from the input device, the command interpreter reads from `SYS$INPUT` to find the data. Thus, you can change the source of your input data by reassigning `SYS$INPUT`.
- `SYS$OUTPUT` is the logical name for the default output file. When you execute a command procedure, `SYS$OUTPUT` is assigned to your terminal, unless you use the `/OUTPUT` qualifier with the `@` (Execute) command. If you submit the procedure as a batch job, `SYS$OUTPUT` is assigned to the LOG file associated with the batch job. The LOG file is written on your login disk area, and when the batch job is completed, the system queues the LOG file to `SYS$PRINT`. You can tell VAX/VMS not to print the LOG file by including the `/NOPRINT` qualifier with the `SUBMIT` command.

You can direct interactive command procedures to accept input from your terminal by temporarily reassigning `SYS$INPUT`. For example, you could create a command procedure that runs a text editor for you, lets you edit as much as necessary, and resumes control when you exit from the editor. The command `ASSIGN/USER__MODE` places a logical name in the process logical name table and removes it after the next image (for example, a DCL command or a program) executing in the process exits.

The following example shows how a command procedure temporarily reassigns `SYS$INPUT` to the terminal:

```
$ ASSIGN/USER_MODE SYS$COMMAND SYS$INPUT
$ EDIT/EDT 'P1',FOR
$ FORTRAN/DEBUG 'P1'
$ LINK/DEBUG 'P1'
$ ASSIGN/USER_MODE SYS$COMMAND SYS$INPUT
$ RUN 'P1'
```

By reassigning `SYS$INPUT` to `SYS$COMMAND`, this command procedure allows you to edit a file at your terminal. When you exit from the editor, `SYS$INPUT` reverts to the command procedure. The command procedure then compiles and links your program, including debugging information. Finally, the command procedure runs your program and, by reassigning `SYS$INPUT` once again, allows you to interact with the debugger.

You can generalize your command procedures by using the INQUIRE, READ, and WRITE commands. INQUIRE writes a prompt string and accepts a value for a symbol. READ accepts data from a specified file. WRITE writes a character string to a specified file.

The following example reworks the preceding one so that the command procedure uses the INQUIRE and WRITE commands to request a file name and write it to the user's terminal before performing each operation.

```
$ INQUIRE FILENAME "File name"
$ ASSIGN/USER_MODE SYS#COMMAND SYS#INPUT
$ EDIT/EDT 'FILENAME',FOR
$ !
$ WRITE SYS$OUTPUT "Compiling ''FILENAME'"
$ FORTRAN/DEBUG 'FILENAME'1 $ !
$ WRITE SYS$OUTPUT "Linking ''FILENAME'"
$ LINK/DEBUG 'FILENAME'
$ !
$ WRITE SYS$OUTPUT "Running debugger with ''FILENAME'"
$ ASSIGN/USER_MODE SYS#COMMAND SYS#INPUT
$ RUN 'FILENAME'
$ EXIT
```

1.6.8 Controlling Command Procedure Execution Flow

Some DCL commands are intended to control the flow of execution in command procedures. The most commonly used flow-control commands are EXIT, GOTO, and IF...THEN.

- EXIT directs the command interpreter to stop processing the current command procedure and resume executing commands interactively (or from the next outermost level of command procedure).
- GOTO tells the command interpreter to change the order in which it processes commands (usually sequential) and to begin processing at a specified label.
- IF...THEN requests evaluation of a condition and performance of an action if the condition holds.

For example:

```
$ INQUIRE CHECK "Y to purge files, E to exit"
$ IF CHECK .EQS. "E" THEN EXIT
$ IF CHECK .NES. "Y" THEN GOTO :BYE
$ PURGE/LOG [KLEE...]
$ BYE:
$ LOGOUT
```

When you execute this command procedure, it prints the following characters on your terminal:

```
Y to purge files, E to exit:
```

If you respond with E, you exit from the procedure immediately. If you respond with Y, the next statement executed is PURGE [KLEE...]. If you respond with anything but E or Y,

including a response of just a RETURN, control is transferred to the first command following the label BYE:. This command procedure is a convenient one to use when you log out, because it reminds you to purge your disk area.

1.6.9 Handling Command Procedure Errors

Two types of errors occur in command procedures: those that result from the command procedure containing errors, and those that result from an attempt by the procedure to perform an operation that fails. VAX/VMS provides the SET VERIFY, ON...THEN, and SET NOON commands and the reserved symbols \$STATUS and \$SEVERITY to help you deal with these errors.

The SET VERIFY command can help you discover and correct command procedure execution errors. When you enter SET VERIFY before executing a command procedure, the system displays the procedure lines as they are executed. Thus, you can see which commands are executed and in what order, and detect errors as they happen.

You have two options in dealing with command procedure operations that fail. You can choose to have a certain command executed whenever an error of a specified severity occurs, using the ON...THEN command. Or you can disable system error checking with SET NOON, and use the VAX/VMS error status information to decide what action to take.

VAX/VMS provides error status information in the reserved global symbols \$STATUS and \$SEVERITY. \$STATUS contains a condition code, while \$SEVERITY contains only the severity code, which is the three low-order bits of \$STATUS. By VAX/VMS convention, an odd condition value signals success and an even value signals failure. The codes and their values are shown in Table 1-5.

Table 1-5: Severity Codes

Value	Severity
0	Warning
1	Success
2	Error
3	Information
4	Severe, or fatal, error

You can use an ON...THEN command to specify execution of a DCL command (such as GOTO) in case of an error. The ON clause names the least severe class of errors for which the DCL command should be executed. The command interpreter executes the THEN clause when errors of the named class or errors in more severe classes occur. The severity levels, in order of increasing severity, are WARNING, ERROR, and SEVERE__ERROR.

For example, if you want your procedure to stop when an error of any severity level occurs, use the following command:

```
# ON WARNING THEN EXIT
```

If you were to specify `ERROR` instead of `WARNING`, the procedure would exit on errors and severe errors, but execution would continue if a warning occurred.

You can use the `SET NOON` command (which is the negation of `SET ON`) to request the system not to check `STATUS`. The command interpreter continues to load the severity code into `STATUS`, but takes no actions based on its value. Thus, you can act on error conditions as you prefer.

When you use `SET NOON`, you usually test `STATUS` with the `IF...THEN` command. Because an odd integer is evaluated as `TRUE` and an even integer, as `FALSE`, you can use the following command to exit when an error of any severity level occurs:

```
# IF .NOT. STATUS THEN EXIT
```

For more information on using `STATUS` and related symbols, and for a full description of error checking in command procedures, see the *Guide to Using DCL and Command Procedures on VAX/VMS*.

1.6.10 Submitting Command Procedures in Batch Mode

The `SUBMIT` command causes the command interpreter to execute your command procedures in batch mode, freeing your terminal for other work. Section 1.6.5 discusses the method of passing parameters to batch jobs. This section demonstrates how you can get `VAX/VMS` to perform the following operations:

- Run your batch job at a specific time
- Tell you when the batch job has completed
- Print or save the `LOG` file

The `/AFTER` qualifier of the `SUBMIT` command tells `VAX/VMS` to wait until the time specified before beginning to execute the batch job. For example, the command

```
# SUBMIT/AFTER=17:00 COMPILES
```

tells `VAX/VMS` to submit the file `COMPILES.COM` to the default batch queue at 5:00 P.M.

The `/NOTIFY` qualifier requests `VAX/VMS` to ring the bell on your terminal and print a message informing you that your batch job has been completed. The message also informs you of the severity level of any error that causes the batch job to be aborted. For example:

```
# SUBMIT/NOTIFY COMPILES
```

The `/NOPRINT` qualifier tells `VAX/VMS` not to queue the `LOG` file for printing. If you do not specify `/NOPRINT`, the `LOG` file is automatically queued to the printer when the batch job exits, and the `LOG` file is deleted when it has been printed. If you wish to print the `LOG` file but do not want it deleted, specify the `/KEEP` qualifier.

The /LOG qualifier allows you to control the creation and location of the LOG file. If you specify /NOLOG, VAX/VMS does not create a LOG file. If you use /LOG=file-spec, VAX/VMS creates the LOG file with the name that you specify. For example, the command

```
$ SUBMIT/NOPRINT/LOG=[ ] COMPILES
```

causes VAX/VMS to create a file called COMPILES.LOG on the current default directory.

1.6.11 Login Command File

VAX/VMS recognizes a special command procedure, called a login command file (or simply a login file), that the system tries to locate and execute every time you log in. When you log in, VAX/VMS looks on your default device and directory for a file called LOGIN.COM. If the file exists, it is executed as an interactive command procedure before the login procedure is completed.

The LOGIN.COM file therefore gives you a way to set up an environment that will be the same for every terminal session. For example, you can define logical names that you use in every session, you can set job and terminal parameters, and so forth. A sample LOGIN.COM file is shown in the following example:

```
$ DEFINE TOOLS USERD:[KLEE.TOOLS]
$ SET TERMINAL /INQUIRE
$ RUN TOOLS:CALENDAR
```

This LOGIN file defines the logical name TOOLS as the subdirectory [KLEE.TOOLS], which contains some commonly used programs such as CALENDAR.EXE.

For more information on DCL command procedures, see the *Guide to Using DCL and Command Procedures on VAX/VMS*.

Chapter 2

Creating and Modifying Programs

The first step in developing a VAX FORTRAN program consists of creating the program's source file. The VAX/VMS EDT text editor can be used to perform this operation. This chapter provides an introduction to the use of EDT.

There are three other sources of information on EDT available to you. The first is the *VAX EDT Reference Manual*. The second is the computer-assisted course titled "Introduction to the EDT Editor" supplied with the VAX/VMS operating system. The third is EDT's help facility, described in Section 2.1.1.

2.1 Introduction to EDT

EDT, the DEC Standard Editor, is an interactive general-purpose text editor. It offers two modes of operation: *line mode editing*, in which operations are performed on single lines of text; and *character mode editing* (also known as *change mode editing*), in which operations are performed on characters and words as well as on lines. Line editing is possible on either hardcopy or video terminals. Character editing, while usable on hardcopy terminals, is most effective on video terminals.

Line editing, with its English-like commands, is easy for the inexperienced user to learn. Character editing, while requiring practice, is also very simple. This makes EDT especially suitable for an inexperienced user who wants to quickly learn how to perform some basic text-handling operations.

EDT also offers many advanced features for experienced users who use it heavily:

- Multiple text buffers. By default, editing operations take place within a single text buffer called MAIN. However, you can maintain an unlimited number of alternate text buffers as "holding areas" for text that you do not necessarily wish to incorporate in the output file.
- Flexible input and output commands. You can copy files into an EDT text buffer after beginning the editing session, and you can output text buffers (or portions of text buffers) to files before ending the session.

- Macro capability. You can create sequences of line editing commands that you invoke with a single command.
- The ability to define keys for custom character editing applications. For example, a keypad key can be defined so that it inserts a specified line of text each time it is pressed. This function is especially useful in programming applications where certain statements may be repeated frequently.

Finally, EDT protects your text. Should your editing session end in an unexpected manner, you can recover all your editing operations by reentering the EDT command line with the /RECOVER qualifier. EDT then “replays” your editing session up to the point of interruption, using the contents of the journal file that it maintained during the lost session.

The following subsections introduce EDT’s help facilities and explain how to invoke and terminate EDT, how to enter and exit editing modes (line mode and character mode), how to protect and recover text, and how to create a new file.

2.1.1 The Help Facilities

EDT offers on-line help in both line mode and character mode.

In line mode, you invoke the help facility by entering the HELP command. Issued without parameters, this command displays information on how to get further help, plus a list of subjects for which help is available. If you enter one of the subjects as a parameter to the HELP command, EDT displays information on that subject, and possibly another list. For example:

```
*HELP DELETE

DELETE

The DELETE (abbreviation: D) command deletes the line specified
.
.
.
Additional information available:

/QUERY
*HELP DELETE /QUERY

DELETE

/QUERY
.
.
.

Q Quit, do not delete any of the remaining
  lines
A All, delete all of the remaining lines

*
```

In character mode, you obtain help by pressing the **HELP** key on your keypad; EDT will display a diagram of the keypad that identifies all of the key functions. You can then obtain help on an individual function by pressing the key that invokes that function. (Figure 2-2 shows the location of the **HELP** key.)

2.1.2 Invoking and Terminating EDT

An editing session begins when you invoke EDT with the **EDIT/EDT** command and ends when you terminate EDT with the **EXIT** or **QUIT** command.

When you start an editing session, you can specify the name of a new file or the name of an existing file. In the former case, you can use your editing session to create a new file with the name that you specified. In the latter case, EDT loads the existing file into its **MAIN** text buffer, and you can then add to or modify the text in the file. EDT does not destroy the contents of the existing file that you are editing; it simply produces a new version, leaving the old version intact.

2.1.2.1 Invoking EDT

To invoke EDT, issue an **EDIT/EDT** command in the format

```
EDIT/EDT[/qualifier...] file-spec
```

Qualifiers

[NO]COMMAND[=file-spec]
[NO]JOURNAL[=file-spec]
[NO]OUTPUT[=file-spec]
[NO]READ
[NO]RECOVER

Defaults

/COMMAND=EDITINI.EDT
/JOURNAL=infile-name.JOU
/OUTPUT=infile-spec
/NOREAD_ONLY
/NORECOVER

where:

file-spec

specifies the file to be created or edited. If the file does not exist, EDT creates it.

EDT does not provide a default file type. If you do not specify one, the file type is null.

/OUTPUT[=file-spec]

supplies an alternate file specification for the output file. By default, EDT creates an output file upon exit that has the same name and type as the input file and a version number of 1 (if the input file does not exist) or 1 higher than the highest existing version (if the input file does exist).

If you specify **/NOOUTPUT**, EDT does not automatically create an output file when you issue the **EXIT** command.

The remaining qualifiers, which describe specialized editor functions, are described elsewhere: the /COMMAND qualifier, in Section 2.4.3; the /JOURNAL, /READ_ONLY, and /RECOVER qualifiers, in Section 2.1.4.

For convenience, you can issue the following command to equate a short command symbol (EDT, in this example) to EDIT/EDT:

```
$ EDT ::= "EDIT/EDT"
```

After you issue this command, the command interpreter will recognize the symbol EDT (or any other symbol you specify) as equivalent to EDIT/EDT.

When you invoke EDT, the response varies depending on whether the file that you specify exists. (Other factors, such as commands contained in a startup command file named EDTINI.EDT (see Section 2.5.3), may further alter the response.) If the file does not exist, EDT so informs you, and prompts you to issue editing commands:

```
$ EDIT/EDT METRIC.FOR
Input file does not exist
[EOB]
*
```

The asterisk (*) is EDT's line editing prompt. When EDT is displaying the asterisk prompt, you can enter any of the commands listed in Table 2-1.

If the file exists, its first line is displayed instead of [EOB]:

```
$ EDIT/EDT METRIC.FOR
 1          PROGRAM METRIC
*
```

NOTE

If you invoke EDT and it does not display an asterisk prompt, you cannot enter line editing commands. This condition can result when the current default directory contains a startup command file named EDTINI.EDT that causes EDT to enter character mode directly. If this happens, you can enter line mode by typing a <CTRL/Z>. You can override any unwanted effects of a startup command file by including the /NOCOMMAND qualifier on the command line.

2.1.2.2 Terminating an EDT Session

Use the EXIT command to terminate EDT and create an output file from the contents of the MAIN text buffer. To override the default output file, you can specify an output file with the EXIT command, as shown in the following example:

```
*EXIT ALTNAME.FOR
_DB1:[PROJECT]ALTNAME.FOR;1 55 lines
$
```

The QUIT command terminates EDT without creating an output file. You can use QUIT if you are simply reading a file without modifying it or if you do not want to save your edits.

The EXIT and QUIT commands are used in both editing modes, and the effects are the same in both modes. In line mode, you simply type the command name after the prompt

(*) and then press <RET>. In character mode, you press the GOLD key; then you press the COMMAND key; then you type the command in response to the prompt; and then you press the ENTER key.

2.1.3 Entering and Exiting Editing Modes

To change from line mode to character mode, use the CHANGE command (abbreviation C). When you issue the CHANGE command, the screen first goes blank and then fills with text. You will find the cursor somewhere on the screen, positioned at the current line or the line you specified with the CHANGE command. (If the buffer is empty, the cursor and [EOB] appear at the top of the screen.)

EDT does not display line numbers while in character mode, although it does continue to assign them as you insert text.

To change from character editing mode to line mode, enter a <CTRL/Z>. This terminates character editing and causes EDT to display the asterisk prompt. You can then perform line editing operations or end the editing session, as appropriate.

2.1.4 Protecting and Recovering Text

Three qualifiers to the EDIT/EDT command allow you to protect files against inadvertent modification and to recover editing operations that have been lost.

The /READ_ONLY qualifier controls whether journaling and the creation of an output file are enabled. (Specifying /READ_ONLY is equivalent to specifying /NOOUTPUT and /NOJOURNAL.) /NOREAD_ONLY, the default, allows EDT to create an output file and a journal file. Use /READ_ONLY in situations where you want to be sure you do not create a modified file, or for reading a file in a directory where you do not have write privileges.

The /JOURNAL qualifier allows you to disable (using /NOJOURNAL) or to specify the name of the journal file that EDT creates to record your editing activity. By default, EDT creates a journal file with the file name of the input file and a file type of JOU. If the editing session ends abnormally, EDT can use the contents of the journal file to re-create the session. If the editing session ends normally (that is, as the result of an EXIT or QUIT command without a /SAVE qualifier), EDT deletes the journal file.

The /RECOVER qualifier causes EDT to use the contents of a journal file to re-create a previous editing session, perhaps one that was lost as the result of an accidental <CTRL/Y> or system problem. If you specify /RECOVER, EDT locates a file with the same name as the input file and a file type of JOU; then it applies all of the editing operations recorded in the journal file to the input file. These operations appear on your terminal as EDT performs them. When EDT has exhausted the contents of the journal file, the activity on the terminal ceases. You can then continue to edit.

Two notes of caution are necessary. First, it is important for the EDIT/EDT command that starts a recovery operation to match exactly the command that started the lost session,

including any special startup command files. The only difference between the two commands should be the /RECOVER qualifier. In particular, the input file must be the same version that you started with at the beginning of the lost session. Second, note that EDT does not necessarily recover your session to the exact point where it was lost. A few keystrokes may be missing, possibly including a partial command sequence. To clear any partial command sequences, press <CTRL/X> before continuing the editing session.

2.1.5 Creating a New File

To create a new file, you issue an EDIT/EDT command that specifies a file that does not currently exist in your directory. In character mode, the designation [EOB] appears on the screen. This indicates that you are currently at end-of-buffer and that any text you insert will be the only text in the buffer. You can then enter as many line of text as you wish. When you have finished entering text, terminate your EDT session as described in Section 2.1.2.2.

In line mode, after EDT responds with the asterisk prompt, issue the INSERT command (abbreviation I) followed by <RET>. The cursor or print head then moves to the right 16 spaces; this space is left by EDT to accommodate line numbers, although none appear at this stage. You can now enter as many lines of text as you wish. When you are finished entering text, terminate the insert with <CTRL/Z>.

In line mode, if you do not want EDT to leave space in front of each line for line numbers, you can issue the SET NONUMBERS command; EDT will then begin each line at the left margin of the terminal. EDT continues to number lines, but does not display the numbers. You can restore the line number display later by issuing a SET NUMBERS command.

2.2 Character Mode Editing

EDT's character mode allows you to perform editing operations at any position in your text instead of line by line. For most applications, especially those requiring extensive modification of existing text, character editing is faster and more straightforward than line editing. When you use character mode on a video terminal, your screen always contains an accurate picture of the area of the file in which you are working. The terminal's cursor shows exactly where you are at all times.

There are two types of character editing: nokeypad and keypad. Nokeypad character editing works on all terminals, including hardcopy terminals. It requires you to enter short commands through the keyboard and terminate each command with a <RET>. Keypad character editing works on the VT50-, VT100-, and VT200-series video terminals and on terminals that are compatible with them. In keypad editing, you request editor functions by pressing keys on the auxiliary keypad; no <RET> is required to terminate the command. Anything you type on the keyboard, including carriage returns, is inserted into the file as text.

This section describes only keypad character editing. To learn about nokeypad character editing, read the *VAX EDT Reference Manual*.

The keypads for the VT52, VT100, and VT200 (and compatible) terminals are different. Therefore, the following description refers to functions rather than to specific keys. It is a good idea to keep a copy of the appropriate keypad diagram handy while you are learning character editing. Figures 2-1, 2-2, and 2-3 contain the keypad diagrams for the VT52, VT100, and VT200, respectively. The numbers or characters shown in the upper right of each key correspond to what you see on the key.

Note that most keys perform two functions. To use the upper of the two functions shown on a key, press the key. To use the lower function, press and release the GOLD key before pressing the function key.

2.2.1 Maneuvering the Cursor

Before performing most character editing operations, you must move the cursor to the location in the file where you wish the operation to take place. There are many ways to move the cursor; experience eventually teaches which is best in a given situation.

The LEFT and RIGHT functions move the cursor one character to the left or right. If the cursor is at the end of a line, the RIGHT function moves it to the beginning of the next line. Conversely, if the cursor is at the beginning of a line, the LEFT function moves it to the end of the previous line.

The UP and DOWN functions move the cursor one line up or down. The column position of the cursor does not change, unless there is no text in the corresponding column above or below (that is, the line that you are moving to is shorter than the line that you are moving from). In the latter case, the cursor moves to the end of the line that is above or below.

The beginning-of-line function, obtained by pressing the BACKSPACE key, moves the cursor to the beginning of the line in which it is positioned. If the cursor is already at the beginning of a line, the function moves it to the beginning of the previous line.

			↑
GOLD	HELP	DEL L UND L	UP REPLACE
7	8	9	↓
PAGE COMMAND	FNDNXT FIND	DEL W UND W	DOWN SECT
4	5	6	→
ADVANCE BOTTOM	BACKUP TOP	DEL C UND C	RIGHT SPECINS
1	2	3	←
WORD CHNGCASE	EOL DEL EOL	CUT PASTE	LEFT APPEND
	0	•	ENTER
LINE OPEN LINE	SELECT RESET	ENTER SUBS	

ZK-020-81

Figure 2-1: VT52 Keypad

↑	↓	←	→
UP	DOWN	LEFT	RIGHT

PF1	PF2	PF3	PF4
GOLD	HELP	FNDNXT FIND	DEL L UND L
7	8	9	→
PAGE COMMAND	SECT FILL	APPEND REPLACE	DEL W UND W
4	5	6	1
ADVANCE BOTTOM	BACKUP TOP	CUT PASTE	DEL C UND C
1	2	3	ENTER
WORD CHNGCASE	EOL DEL EOL	CHAR SPECINS	ENTER SUBS
	0	•	
LINE OPEN LINE	SELECT RESET		

ZK-021-81

Figure 2-2: VT100 Keypad

VT200

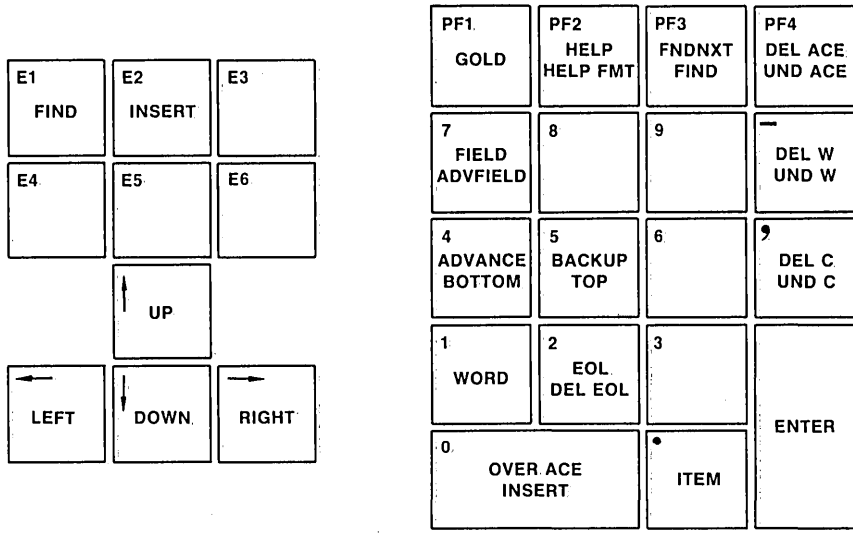


Figure 2-3: VT200 Keypad

The TOP and BOTTOM functions move the cursor to the beginning and end of the buffer, respectively.

All the remaining cursor movement functions depend in part on the ADVANCE and BACKUP functions. The ADVANCE function causes subsequent cursor movement to occur in the forward direction, that is, toward the end of the buffer. The BACKUP function causes subsequent cursor movement to occur in the backward direction, toward the beginning of the buffer. When character editing begins, cursor movement is forward, until reversed by the BACKUP function.

The following functions depend on the current direction established by ADVANCE and BACKUP:

- The CHAR function moves the cursor one character.
- The WORD function moves the cursor to the beginning of the next or previous word (the end-of-line character is considered a word).
- The LINE function moves the cursor to the beginning of the next line, if the current direction is forward. If backward, the LINE function moves the cursor to the beginning of the line in which the cursor is positioned, or, if the cursor is at the beginning of a line, to the beginning of the previous line.
- The EOL (for end-of-line) function moves the cursor to the next or previous end-of-line character.

- The SECT (for section) function moves the cursor one 16-line section.
- The PAGE function moves the cursor to the next or previous page mark (by default, a form feed).

All of these cursor movement functions can be combined with a repeat count, causing the function to be repeated a specified number of times. To enter a repeat count, press the GOLD key, then type in the count on the keyboard (not keypad) number keys, then type in the function to be repeated. As you enter the repeat count, the numbers appear on the screen below the area reserved for text. The numbers disappear as soon as you enter the function.

You can also use FIND and FNDNXT (for find next) to move the cursor to a certain string. To find a string, enter the FIND function. EDT prompts you for a search string. Type the search string without delimiters, and terminate it with either the ADVANCE or BACKUP function to determine the direction of search. EDT moves the cursor to the beginning of the search string. If the search string is not found, EDT issues a message and does not move the cursor.

The FNDNXT function finds the next occurrence of the current search string in the current direction. The current search string is the last string you entered with the FIND function.

Note that you can locate strings that include carriage returns with the FIND function. Simply enter the carriage return as part of the search string. The carriage return does not terminate the search string; you do that with the ADVANCE or BACKUP function. EDT echoes a carriage return in a search string as ^M.

2.2.2 Inserting New Text

Once the cursor is positioned, you can insert text in front of it simply by typing the text on the keyboard. No command is required and whatever you type becomes part of the file. Your insertion appears on the screen as you type it, and the surrounding text moves as necessary to accommodate it.

When you insert text at the beginning or in the middle of a line, the end of the line may disappear off the edge of the screen. The text is not lost, however: if you enter a carriage return in the text you are typing, the text appears on the next line. To avoid this problem, you can use the OPEN LINE function. When the cursor is at the beginning of a line, OPEN LINE provides a blank line above that line, and positions the cursor at the beginning of the blank line.

As you type new text, you may notice errors in surrounding text. You can move the cursor to these errors and correct them at any time, and then move the cursor back and continue to insert text.

2.2.3 Deleting and Undeleting Text

EDT character editing provides several methods of deleting text in units of varying sizes. EDT also maintains three buffers to contain text that has been deleted. The character

buffer contains the last character deleted; the word buffer contains the last word deleted; and the line buffer contains the last line deleted. You can insert the contents of each of these three buffers at the cursor position by using the UND C, UND W, and UND L functions, respectively. There is no limit to the time or number of operations between a delete operation and the undelete operation that reinserts the deleted text. Furthermore, you can undelete one unit of text as many times as you wish, and at any locations you wish.

The DEL C (for character) function deletes the character at which the cursor is positioned, and moves the cursor to the next character. The DELETE key on the keyboard deletes the character before the cursor position (the last character typed, if you are inserting text) and does not change the cursor position. Both of these functions move the deleted character into the character buffer, from which it can be retrieved by using the UND C function.

The DEL W (for word) function deletes from the current cursor position to (but not including) the first character of the next word. The LINE FEED key on the keyboard deletes from (but not including) the cursor position back to the first character of the current word. Both of these functions move the deleted text into the word buffer, from which it can be retrieved by using the UND W function.

The DEL L (for line) function deletes from the cursor position through the next end-of-line character. The DEL EOL (for end-of-line) function is similar, except that it does not delete the end-of-line character. Typing <CTRL/U> deletes from (but not including) the cursor position to the beginning of the current line. All of these functions move the deleted text into the line buffer, from which it can be retrieved by using the UND L function.

2.2.4 Moving Text

Character editing provides two basic methods of moving text. The first is available through the three undelete functions. You can delete a unit of text from one location, move the cursor to another location, and undelete the text there. However, this method is only effective for units that can be deleted by the various functions described in Section 2.2.3. To move larger or more precise blocks of text, use CUT and PASTE. These two functions allow you to “cut” any amount of contiguous text from one location and “paste” it elsewhere.

The first step is defining the text to be moved. To do this, move the cursor to either the beginning or end of the text and enter the SELECT function. Then move the cursor to the other extremity of the text. In so doing, you create a select range, that is, all of the text between the cursor position and the position at which you entered the SELECT function. On VT100 and VT200 terminals, EDT highlights the select range with reverse video. If you make a mistake while you are defining the select range, enter the RESET function to cancel the select range currently in effect.

Once you have defined the select range, enter the CUT function. The text within the select range disappears. (EDT moves it into a text buffer named PASTE.) Move the cursor to the position at which the text is desired, and enter the PASTE function. The text appears at the cursor position.

You can paste the cut text in as many locations as required. Specifically, you can paste the text as soon as you cut it, then move the cursor and paste the text again. This is in effect a copy operation.

Each CUT operation destroys the previous contents of the PASTE buffer and replaces them with the select range. To add the select range to the contents of the PASTE buffer, use the APPEND function.

The PASTE buffer is an ordinary EDT text buffer. You can edit within it, load it from a file with the INCLUDE command, or create a file from its contents with the WRITE command.

2.3 Line Mode Editing

To edit an existing file in your directory, issue an EDIT/EDT command that specifies its name. (To edit a file from a directory other than your own, see Section 2.3.10). EDT displays the first line in the file, as shown in the following example:

```
$ EDIT/EDT EXAMPLE.TXT
 1          This is the first line of EXAMPLE.TXT
*
```

The number 1 to the left of the line is the line number. It is not part of the file. The file starts with the word “This.”

The line displayed is the current line. EDT uses the current line as the default in many of its operations. For example, an INSERT command that does not specify a range causes EDT to insert text in front of the current line.

The concept of “range” is central to all EDT line editing operations. The next section describes ways of specifying range. The sections that follow describe the most common and useful line editing operations.

2.3.1 Line Editing Command Summary

When you invoke EDT, and throughout your editing session, EDT prompts you to enter line editing commands by displaying an asterisk. For example:

```
$ EDIT/EDT METRIC.FOR
 1          PROGRAM METRIC
*
```

Table 2-1 describes briefly (in alphabetical order) the most useful commands that you can enter in response to the line editing prompt (*). Each command has a smallest acceptable abbreviation, shown in bold type in the table.

All line editing commands are terminated with a <RET>. Most of the commands allow or require you to specify a range or ranges; the range specification tells EDT where the action of the command should take place. Section 2.3.2 summarizes range specifications, and the command examples show various ways of specifying a range.

Table 2-1: Summary of Line Editing Commands

Command	Function
CHANGE [range]	Invokes character mode editing for specified buffer
CLEAR	Deletes the contents of a text buffer
COPY [range1] TO [range2] [/QUERY]	Copies lines specified by range1 to a location in an EDT buffer specified by range2; does not delete lines from original location
DEFINE { KEY MACRO }	Defines a new or revised key function for character mode editing, or defines a macro name
DELETE [range] [/QUERY]	Deletes a specified line or lines
EXIT [file-spec]	Terminates EDT, saving the contents of the text buffer MAIN as the output file
FILL [range]	Reformats a block of text, filling lines with the maximum number of full words without exceeding the right margin
FIND range	Establishes the first line in range as the current line
HELP [topic ...]	Displays information on the specified EDT command or function
INCLUDE file-spec [range]	Copies an external file to a location in a text buffer specified by range
INSERT [range]	Opens a text buffer for the insertion of text at the location specified by range
MOVE [range1] TO [range2] [/QUERY]	Moves lines specified by range1 to the location specified by range2, deleting the lines from the source location
PRINT file-spec [range]	Creates a listing file with the specified file name
QUIT [/SAVE]	Terminates EDT without creating an output file, optionally saving the journal file
REPLACE [range]	Deletes specified lines from a text buffer and leaves the buffer open for insertion of text
RESEQUENCE [range]	Assigns new line numbers to a range of lines
SET [parameter]	Sets a variety of editor operating parameters
SET [(NO)NUMBER]	Enables/disables the display of line numbers
SHOW [parameter]	Displays specified editor operating parameters
SUBSTITUTE /string1/string2/[range] [/QUERY]	Replaces string1 with string 2, either in the current line or in the specified range

Table 2-1 (Cont.): Summary of Line Editing Commands

Command	Function
<code>[SUBSTITUTE] NEXT</code> <code>[/string1/string2]</code>	Replaces string1 with string2, based either on the strings specified or on the previous SUBSTITUTE command
<code>TAB ADJUST [-]n [range]</code>	Shifts each line in a range of lines by a specified number of logical tab stops
<code>[TYPE] [range]</code>	Displays specified lines and makes the first line in range the current line; the default command
<code>WRITE file-spec [range]</code>	Moves a copy of specified text from a buffer to a file

2.3.2 Specifying Line Ranges

A range is the line or lines on which EDT performs an operation. A range specification is a description of a range in terms that EDT can understand. All the line editing commands (except SUBSTITUTE NEXT) described in the sections that follow accept one or more range specifications, although many do not require one.

The simplest range specification identifies a single line of text. A line can be located by its position in the file relative to the current line, by a text string that it must contain, or by its line number.

When you insert lines of text in a new file, or when EDT loads an existing file into its MAIN buffer, each line of the file receives a number. The numbering starts with 1 and is incremented by 1s. If you insert lines of text between existing lines, EDT numbers the new lines using appropriate decimal increments. This technique ensures enough unique line numbers to cover any reasonable editing operation. EDT displays the line numbers whenever it displays text, unless you have issued the SET NONUMBERS command. In that case, EDT does not display line numbers, but it does continue to assign them.

Single-line range specifications are listed in Table 2-2; examples appear below the table.

Table 2-2: Single-Line Range Specifications

Specification	Meaning
.	The current line
number	The line specified by the number
'string' or "string"	The next line containing the string you specify
-'string' or -"string"	The preceding line containing the string you specify
[range] { + - } [number]	The line that is the specified number of lines after (or before, if minus) the single line specified by range (range defaults to the current line; number defaults to 1)
BEGIN	The first line in the text buffer
END	An empty line (designated by [EOB]) following the last line of text in the text buffer

Examples:

Specification	Meaning
20.6	The line numbered 20.6
"EQUIVALENCE"	The next line that contains the string <i>EQUIVALENCE</i>
"-D__FLOATING COMPLEX"	The first preceding line that contains the string <i>D__FLOATING COMPLEX</i>
-6	The line six lines before the current line
"SUBROUTINE"+4	The line four lines after the line that contains the string <i>SUBROUTINE</i>

When EDT searches for a string, the case of the search string need not match the case of the target. For example, *record* is a match for *RECORD* or *Record*. This condition is the default; you can change it with the SET SEARCH command.

There are several methods available for specifying a range of more than one line. They are listed in Table 2-3; examples appear below the table.

Table 2-3: Multiple-Line Range Specifications

Specification	Meaning
[range1] { : THRU } [range2]	The set of lines from range1 through range2, which are single line range specifications (the default for both range1 and range2 is the current line)
[range] { # FOR } number	The specified number of lines beginning with the single line specified by range (the default for range is the current line)
BEFORE	All lines in the buffer that precede the current line
REST	The current line and all lines in the buffer that follow it
WHOLE	The entire buffer
range, range... or range AND range AND...	All lines specified by each single line range
[range] ALL 'string'	All lines in the range containing the specified string (the default for range is the entire buffer)

Examples:

Specification	Meaning
2:6.5	Lines 2 through 6.5, inclusive
'STRUCTURE' #5	The line containing the string <i>STRUCTURE</i> and the four lines following it, for a total of five lines
.-10:.	The line 10 lines before the current line through the current line, inclusive
10:50 ALL 'READ'	All lines from line 10 through line 50 that contain the string <i>READ</i>

Most range specifications can be combined with a text buffer specification. During your editing session, you may wish to hold and edit text in buffers other than MAIN. To create and gain access to alternate buffers, include the name of the buffer in a range specification, using the following syntax:

```
=buffer [range]
or
BUFFER buffer [range]
```

In this syntax, “buffer” stands for the name of the buffer. It can be from 1 to 30 alphanumeric characters, but it must start with an alphabetic character. If you include a range of lines following the buffer name, you specify the range within the named buffer. If you omit the range specification, you specify either the entire named buffer or its first line, depending on context.

The following examples show buffer specifications in use.

Specification	Meaning
=PROG1	The entire contents of the text buffer named PROG1, or (for commands requiring a single-line range specification) its first line
=INC 'STRUCTURE': 'END STRUCTURE'	The lines that contain the strings <i>STRUCTURE</i> and <i>END STRUCTURE</i> in the text buffer named INC, and all lines between
=COM ALL 'LOGICAL'	All lines that contain the string <i>LOGICAL</i> in the buffer named COM

2.3.3 Displaying Lines of Text

The TYPE command and a RETURN in response to the asterisk prompt are the two methods of displaying text.

TYPE Command

The TYPE command, followed by a range, causes EDT to display the line or lines in the range and resets the current line to the first (or only) line displayed. The word TYPE (abbreviation T) is optional; it does not need to be entered. For example:

```
*T 1:3
  1          This is the first line of EXAMPLE.TXT
  2          This is the second line of EXAMPLE.TXT
  3          This is the third line of EXAMPLE.TXT
*4#2
  4          This is the fourth line of EXAMPLE.TXT
  5          This is the fifth line of EXAMPLE.TXT
*
```

If you do not include the word TYPE and the range specification begins with an alphabetic character (such as WHOLE or REST), you must precede it with a percent sign (%). Otherwise, EDT tries to interpret the range specification as a command. For example:

```
*REST
^
Unrecognized command
*%REST
  4          This is the fourth line of EXAMPLE.TXT
  5          This is the fifth line of EXAMPLE.TXT
  6          This is the sixth line of EXAMPLE.TXT
  7          This is the seventh line of EXAMPLE.TXT
*
```

To cancel any form of type command while text is being displayed, enter a <CTRL/C>. This operation does not cause repositioning; the first line displayed remains as the current line.

RETURN

A RETURN in response to the asterisk prompt displays the line following the current line and sets the current line to the displayed line. A series of returns, therefore, displays successive lines and sets the current line to the displayed line each time. This is an easy way to work through a file line by line. For example:

```
*<RET>
  5          This is the fifth line of EXAMPLE.TXT
*<RET>
  6          This is the sixth line of EXAMPLE.TXT
*
```

2.3.4 Maneuvering in a File

The FIND command (abbreviation F) locates a specified line without displaying it. It is useful for setting the current line to the top of a large block of text that would be cumbersome to display on the terminal. For example, each of the following commands resets the current line to the top of the MAIN text buffer:

```
*=MAIN
*F =MAIN
```

However, the first command (an implied TYPE command) displays the entire contents of the MAIN text buffer. The second command just sets the current line and displays an asterisk prompt.

If you specify a range that EDT cannot locate, EDT issues a message and does not change the current line setting.

2.3.5 Inserting New Text

The procedure for inserting new text in a buffer already containing text is exactly the same as that for inserting text in an empty buffer (see Section 2.1.5), except that you can control where the text goes by including a range specification with the INSERT command. The lines you insert are placed in front of the line you specify. If you specify multiple lines, the insert goes in front of the first line in the range. If you omit the range specification, the insert goes in front of the current line.

In the following example, the INSERT command causes EDT to insert text in front of line 5 in the current buffer. Then, the range specification (an implied TYPE command) causes EDT to display lines 4 through 6, showing the result of the insertion.

```

*I 5
    First insert line
    Second insert line
    Third insert line
    <CTRL/Z> ^Z
* 4:6
4      This is the fourth line of EXAMPLE.TXT
4.1    First insert line
4.2    Second insert line
4.3    Third insert line
5      This is the fifth line of EXAMPLE.TXT
6      This is the sixth line of EXAMPLE.TXT
*

```

NOTE

EDT, which inserts text in front of the current line, is different from many other text editors that insert text following the current line.

2.3.6 Deleting and Replacing Text

Use the DELETE command (abbreviation D) to delete a specified range. If you omit the range, the DELETE command deletes the current line. After a delete operation, EDT displays the line following the last line deleted; this is the new current line. For example:

```

*D 4.1#2
2 lines deleted
    4.3      Third insert line
*D
1 line deleted
    5      This is the fifth line of EXAMPLE.TXT
*

```

The /QUERY qualifier to the DELETE command causes EDT to prompt you before deleting each line of the range. The prompt is a question mark (?). You can respond to the prompt in one of four ways:

Y (yes)	Delete this line
N (no)	Do not delete this line
A (all)	Delete all remaining lines in the specified range
Q (quit)	Quit the delete operation

The REPLACE command (abbreviation R) deletes a specified range and allows you to insert lines to replace those deleted. You terminate the insertion with a <CTRL/Z>, just as with the INSERT command.

2.3.7 Moving Text

The COPY and MOVE commands (abbreviations CO and M, respectively) allow you to move one or more lines of text from one place in the buffer to another, or from one buffer to another. The effect of these commands is similar; the only difference is that the COPY command does not delete the text from its original location, whereas the MOVE command does.

The following example illustrates both commands, as well as alternative ways of specifying a range:

```
*%WHOLE
 1      This is the first line of EXAMPLE.TXT
 2      This is the second line of EXAMPLE.TXT
 3      This is the third line of EXAMPLE.TXT
 4      This is the fourth line of EXAMPLE.TXT
 5      This is the fifth line of EXAMPLE.TXT
 6      This is the sixth line of EXAMPLE.TXT
 7      This is the seventh line of EXAMPLE.TXT

*COPY 1:3 TO 'SIXTH'
3 lines copied
*5:6
 5      This is the fifth line of EXAMPLE.TXT
 5.1    This is the first line of EXAMPLE.TXT
 5.2    This is the second line of EXAMPLE.TXT
 5.3    This is the third line of EXAMPLE.TXT
 6      This is the sixth line of EXAMPLE.TXT

*M 5.1#3 TO BEGIN
3 lines moved
*%WH
 0.1    This is the first line of EXAMPLE.TXT
 0.2    This is the second line of EXAMPLE.TXT
 0.3    This is the third line of EXAMPLE.TXT
 1      This is the first line of EXAMPLE.TXT
 2      This is the second line of EXAMPLE.TXT
 3      This is the third line of EXAMPLE.TXT
 4      This is the fourth line of EXAMPLE.TXT
 5      This is the fifth line of EXAMPLE.TXT
 6      This is the sixth line of EXAMPLE.TXT
 7      This is the seventh line of EXAMPLE.TXT

*
```

The /QUERY qualifier to either COPY or MOVE causes EDT to prompt you before copying or moving each line of the range. It operates the same way as the /QUERY qualifier to DELETE (see Section 2.3.6).

2.3.8 Substituting Text

Two commands, SUBSTITUTE and SUBSTITUTE NEXT, substitute one string for another within a line or lines. These are the only line editing commands that can alter text within a line, as opposed to changing the entire line. The SUBSTITUTE command (abbreviated S), substitutes one string for another within a line or lines. The SUBSTITUTE NEXT command (abbreviated SN), substitutes one string for another within a line or lines, starting from the next line.

viation S) operates on the current line or on a specified range; the SUBSTITUTE NEXT command (abbreviation N) makes a substitution at the next opportunity within the buffer.

The format of the SUBSTITUTE command is:

```
SUBSTITUTE /string1/string2/[range] [/QUERY]
```

The command finds string1 and substitutes string2 for it. If you do not specify a range, the substitution takes place in the current line. If you do, the command makes every substitution within the range. The following example illustrates the command first without and then with a range specified:

```
*1
  1          This is the first line of EXAMPLE.TXT
*S /first/1st/
  1          This is the 1st line of EXAMPLE.TXT
1 substitution
*S /of/in/4:6
  4          This is the fourth line in EXAMPLE.TXT
  5          This is the fifth line in EXAMPLE.TXT
  6          This is the sixth line in EXAMPLE.TXT
3 substitutions
*
```

Slashes (/) are not the only characters you can use to delimit string1 and string2. Any nonalphanumeric character will work, as long as the delimiters are matched and do not occur in either string. For example, the following command substitutes the string A/3 for A/2 in the current line, using dollar signs (\$) as delimiters:

```
*S $A/2$A/3$
  25          SIZE = A/3
1 substitution
*
```

The /QUERY qualifier to SUBSTITUTE causes EDT to prompt you before making each substitution. It operates the same way as the /QUERY qualifier to DELETE (see Section 2.3.6).

The SUBSTITUTE NEXT command (abbreviation N) substitutes for the next occurrence of string1 that it finds in the buffer. If you specify neither string1 nor string2, the command takes the values of both strings from the last SUBSTITUTE command you issued. For example:

```
*N / in/ of/
  4          This is the fourth line of EXAMPLE.TXT
*N
  5          This is the fifth line of EXAMPLE.TXT
*
```

2.3.9 Input From and Output To Files

Two EDT commands, INCLUDE and WRITE, allow you to incorporate text from files and output text to files during your editing session.

INCLUDE Command

The INCLUDE command (abbreviation INC) incorporates the contents of a file at a specified location in a text buffer. If you do not want the entire file incorporated in the MAIN text buffer, you can specify an alternate buffer as the range and then copy the desired portions of the file to their proper places in MAIN. For example:

```
*INC SBRTNES.FOR =SUBS  
*
```

This command creates a buffer called SUBS and fills it with the contents of the file SBRTNES.FOR from the EDT default directory (that is, the directory of the input file given with the EDIT/EDT command).

WRITE Command

The WRITE command (abbreviation WR) creates a file by copying the contents of a specified range in a text buffer. The text is not deleted from the text buffer and EDT does not terminate following the operation. If you do not specify a range with the command, EDT outputs the entire contents of the current text buffer. The following example shows the command used with a range:

```
*WR ROUTINE1.FOR =SUBS 'SUBROUTINE': 'END'  
_DB1:[PROJECT]ROUTINE1.FOR;1 45 lines  
*
```

This command creates the file ROUTINE1.FOR from the lines that contain the strings *SUBROUTINE* and *END* in the buffer named SUBS, and all lines in between.

Unless you include a directory in the file specification, WRITE always creates the file in your current default directory. This is true even if the input and output files are in another directory.

2.3.10 Editing a File From Another Directory

You can edit a file that exists in another directory and use the /OUTPUT qualifier to EDIT/EDT to direct the output file to your directory. However, EDT uses the directory of the input file that you specify in the EDIT/EDT command line as its default directory. This default has the following effects:

- EDT attempts to create its journal file in its default directory, that is, in the other directory. If you do not have the privilege to do this, EDT issues an error message and terminates. You should instead use the /JOURNAL qualifier to place the journal file in your directory. (See Section 2.1.4 for a description of the journal file and /JOURNAL.)
- If you issue an INCLUDE command and do not specify a directory, EDT attempts to locate the file in its default directory, that is, in the other directory. To specify a file in your own directory, use a directory specification with INCLUDE.

In the following example, a user with the directory [WYLBUR] edits a file from the directory [PROJECT]:

```
$ EDIT/EDT [PROJECT]DATADEF.FOR -  
_ $ /OUTPUT=[WYLBUR] /JOURNAL=[WYLBUR]  
:  
*INCLUDE [WYLBUR]ENTRIES.FOR
```

The input file for this editing session is [PROJECT]DATADEF.FOR; the output file is [WYLBUR]DATADEF.FOR. The INCLUDE command incorporates a file from directory [WYLBUR]. If the INCLUDE command had not specified a directory, EDT would have looked for the file [PROJECT]ENTRIES.FOR.

2.4 EDT Aids for the Programmer

In addition to the general-purpose editing operations discussed in Section 2.1, EDT provides some advanced functions that are especially useful for programming. The following sections introduce some of these.

2.4.1 Structured Tabs

Although FORTRAN is a free-form language, in which excess spaces and tabs have no significance, it is common practice to indent lines to indicate the relationship of statements. It is laborious to enter repeatedly the correct combination of tabs and spaces to achieve the desired indentation. EDT solves this problem by providing a system of structured tabs in character mode editing. While you are inserting text, a depression of the tab key inserts the correct combination of tabs and spaces to bring the cursor to the desired column. When you need to begin lines at a different column, you can increase or decrease the indentation level to move the starting column to the right or left, respectively, by a preset increment.

To use the structured tab feature, follow these steps:

1. While in line mode, set the increment between tabs by issuing the SET TAB command with a suitable value. For example:

```
*SET TAB 4  
*
```

At this point, the first <TAB> on a line (while in character mode) positions the cursor at column 5. Subsequent tab stops are at the normal locations.

2. When you want to change the indentation level, use <CTRL/E> or <CTRL/D>. Each depression of <CTRL/E> increases the indentation by one increment; the first tab stop is n spaces further to the right, where n is the number you gave with the SET TAB command. Pressing <CTRL/D> decreases the indentation level.

3. If you want to set the indentation level to correspond to a given column, position the cursor at that column and press <CTRL/A>. The column must be at an even multiple of n spaces from the left edge of the screen.
4. If you want to change the indentation of a block of lines, first define a select range that includes the lines to be shifted. (To define a select range, position the cursor at one end of the block of lines, enter the SELECT function, and then position the cursor at the other end.) Then enter a repeat count (the GOLD key followed by a number typed on the keyboard) to indicate how many units of n spaces the lines should be shifted. A positive repeat count shifts the lines to the right; a negative repeat count shifts the lines to the left. Finally, press <CTRL/T>.

Before you enter <CTRL/T> at the EDT command level, note that you must disable the DCL <CTRL/T> if you want the EDT <CTRL/T> to take effect. You establish the DCL <CTRL/T> mode with the SET [NO]CONTROL=T command.

2.4.2 Special-Purpose Key Definitions

EDT allows you to redefine the functions invoked by all the keys on the auxiliary keypad and many control characters as well. There are two ways to redefine a key's function:

- While in character mode, press <CTRL/K>. EDT prompts you to press the key you wish to define. Once you have pressed the key, EDT prompts you to enter the new function. You can do this either by typing the nokeypad commands that make up the function, or by pressing the keypad keys that correspond to the functions you require. You must follow the function specification with a period. The ENTER function terminates a definition of this type.
- While in line mode, issue the DEFINE KEY command. You define the new function to perform as a string of nokeypad character editing commands, followed by a period. The string and period must be enclosed in quotes.

Key redefinition requires a good grasp of nokeypad character editing syntax, as well as a good deal of practice. The EDT help facility (particularly HELP DEFINE KEY and HELP CHANGE SUBCOMMANDS) and the *VAX EDT Reference Manual* are good sources of information. However, this section describes one common application: the redefinition of a key to insert a string of text.

While writing a program, you may find that you are typing the same group of words over and over. For example, you might get tired of typing *CHARACTER**. In character mode, follow this procedure to define a key to insert the string *CHARACTER**:

1. Press <CTRL/K>. EDT prompts you as follows:

```
Press the key you wish to define
```

2. Select a function that you do not use often, for example, SPECINS. You might also select a control character. Enter the function or control character. EDT then prompts you as follows:

```
Now enter the definition terminated by ENTER
```

3. Type the following:

```
iCHARACTER*<CTRL/Z>.
```

(The period is required syntax.)

4. Press ENTER to terminate the definition procedure.

For the remainder of the editing session, the key that used to invoke the SPECINS function will instead insert the string *CHARACTER** at the cursor position.

In line mode, you can redefine a key by using the DEFINE KEY command. To identify a keypad key in the command, you use a number. You can find out which numbers are assigned to which keys by issuing the command HELP DEFINE KEY VT52 or HELP DEFINE KEY VT100. These commands display the numbers assigned to keypad keys on the respective terminals.

Next, you issue a DEFINE KEY command, specifying the key and the function you wish the key to perform. The following example redefines the SPECINS function (GOLD/3 on a VT100) to insert the string *CHARACTER**:

```
*DEFINE KEY GOLD 3 AS "iCHARACTER*^Z."  
*
```

The quotes and period are required syntax. The ^Z is *not* a <CTRL/Z>, but a circumflex followed by a Z; it indicates the end of an inserted string. or the remainder of the editing session, GOLD/3 will insert the string *CHARACTER** at the cursor position.

The preceding examples represent only a small fraction of the capabilities of key redefinition. With practice, you can create powerful custom functions that can save you a great deal of time. You may want to store these functions in a startup command file so that you will not have to define them each time you begin an editing session. The next section describes startup command files.

2.4.3 Startup Command Files

When you invoke EDT, it searches your current default directory for a file named EDTINI.EDT. If EDT finds such a file, it executes the line editing commands contained in the file before turning control over to you. This function allows you to customize EDT to suit your needs. Some of the commands that a startup command file might contain are:

- **DEFINE KEY.** These commands redefine the function invoked by a keypad key or control character while in character mode. (See Section 2.4.2.)
- **DEFINE MACRO.** These commands associate a name with a sequence of line-editing commands stored in a text buffer. You can then invoke the sequence by entering the macro name in response to the line-editing asterisk prompt.
- **INCLUDE.** These commands bring text from a file into a text buffer. You might use them to load macros into a buffer, or to fill a buffer with text that you often use. (See Section 2.3.9.)

- **SET.** These commands establish EDT operating parameters. Particularly useful are **SET TAB**, which establishes the increment for structured tabs, and **SET MODE CHANGE**, which causes EDT to enter directly into character mode. (Section 2.4.1 describes the use of structured tabs.)

You can use the **/COMMAND** qualifier to the **EDIT/EDT** command to cause EDT to search for a file other than **EDTINI.EDT**. This means that you can have several startup command files, each designed for a particular application. You may want to include a command in your login command procedure file (see Section 1.6.11) to equate a short mnemonic to an **EDIT/EDT** command that invokes a special startup command file. For example, if you have the following line in your login command file:

```
EDP ::= "EDIT/EDT/COMMAND=FORT,EDT"
```

then the comand

```
EDP METRIC,FOR
```

invokes EDT with the startup command file **FORT.EDT** to edit the file **METRIC.FOR**.

Chapter 3

Compiling FORTRAN Programs

This chapter describes how to use the FORTRAN command to compile your source programs into object modules. The following topics are discussed:

- The functions of the compiler (Section 3.1)
- The syntax of the FORTRAN command and its qualifiers (Section 3.2)
- The use of text libraries (Section 3.3)
- The Common Data Dictionary (CDD) (Section 3.4)
- Compilation control statements (Section 3.5)
- Compiler diagnostic messages and error conditions (Section 3.6)

3.1 Functions of the Compiler

The primary functions of the VAX FORTRAN compiler are as follows:

- To verify the FORTRAN source statements and to issue messages if there are any errors
- To generate machine language instructions from the source statements of the FORTRAN program
- To group these instructions into an object module for the linker

When the compiler creates an object file, it provides the linker with the following information:

- The program unit name. This is taken from the name specified in the PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA statement in the source program. If a program unit does not have any of these statements, the source file name, with "\$MAIN" (or "\$DATA", for block data subprograms) appended, is used.
- A list of all entry points and common block names that are declared in the program unit. The linker uses this information when it binds two or more program units together and must resolve references to the same names in the program units.

- Traceback information. This is used by the system default condition handler when an error occurs that is not handled by the program itself. The traceback information permits the default handler to display a list of the active program units in the order of activation, which aids program debugging.
- If specifically requested (with the /DEBUG qualifier), a symbol table. A symbol table lists the names of all external and internal variables within a module, with definitions of their locations. The table is of primary use in program debugging.

The linker is described in Chapter 4.

3.2 The FORTRAN Command

The FORTRAN command initiates compilation of a source program.

The command has the form:

```
$ FORTRAN[/qualifiers] file-spec-list[/qualifiers]
```

where:

/qualifiers

Indicates either special actions to be performed by the compiler or special properties of input or output files.

file-spec-list

Specifies the source file(s) containing the program unit(s) to be compiled. You can specify more than one source file. If source file specifications are separated by commas, the programs are compiled separately. If source file specifications are separated by plus signs, the files are concatenated and compiled as one program.

In interactive mode, you can also enter the file specification on a separate line by typing the command FORTRAN, followed by a carriage return. The system responds with the prompt:

```
_File:
```

Type the file specification immediately after the prompt and then type RETURN.

3.2.1 Specifying Input Files

In specifying a list of input files on a FORTRAN command, you can use abbreviated file specifications for those files that share common device names, directory names, or file names. The system applies temporary file specification defaults to those files with incomplete specifications. The defaults applied to an incomplete file specification are based on the previous device name, directory name, or file name encountered in the list.

For example, assume that the current default device and directory name are `USR2:[MONROE]`. The following FORTRAN command shows how temporary defaults are applied to a list of file specifications:

```
$ FORTRAN USR1:[ADAMS]TEST1,TEST2,[JACKSON]SUMMARY,USR3:[FINAL]
```

The preceding FORTRAN command compiles the following files:

```
USR1:[ADAMS]TEST1.FOR  
USR1:[ADAMS]TEST2.FOR  
USR1:[JACKSON]SUMMARY.FOR  
USR3:[FINAL]SUMMARY.FOR
```

To override a temporary default with your current default directory, specify the directory as a null value. For example:

```
$ FORTRAN [ALPHA]TEST1, []TEST2
```

In this case, the empty brackets indicate that the compiler is to use your current default directory to locate `TEST2`.

You must use `/LIBRARY` qualifiers in your FORTRAN command if text libraries are accessed by programs in the source files that you specify. The `LIBRARY` qualifier is discussed at length in Section 3.3.3.

3.2.2 Specifying Output Files

The output produced by the compiler includes the object and listing files. You can control the production of these files by using the appropriate qualifiers in the FORTRAN command.

The compiler generates an object file by default. In interactive mode, the compiler does not generate listing files; you must use the `/LIST` qualifier to generate the listing file. In batch mode, however, the compiler generates a listing file by default. To suppress it, use the `/NOLIST` qualifier.

During the early stages of program development, you may find it helpful to use the `/NOOBJECT` qualifier to suppress the production of object files until your source program compiles without errors. If you do not specify `/NOOBJECT`, the compiler generates object files as follows:

- If you specify one source file, one object file is generated.
- If you specify multiple source files, separated by commas, each source file is compiled separately and an object file is generated for each source file.
- If you specify multiple source files, separated by plus signs, the source files are concatenated and compiled, and one object file is generated.

You can use both commas and plus signs in the same command line to produce different combinations of concatenated and separate object files (see Example 4).

To produce an object file with an explicit file specification, you must use the /OBJECT qualifier, in the form /OBJECT=file-spec (see Section 3.2.3.14). Otherwise, the object file has the name of its corresponding source file and a file type of OBJ. By default, the object file produced from concatenated source files has the name of the first source file. All other file specification fields (node, device, directory, and version) assume the default values.

The following examples show a variety of FORTRAN commands. Each command is followed by a description of the output file(s) it produces.

1. \$ FORTRAN/LIST AAA,BBB,CCC

Source files AAA.FOR, BBB.FOR, and CCC.FOR are compiled as separate files, producing object files named AAA.OBJ, BBB.OBJ, and CCC.OBJ; and listing files named AAA.LIS, BBB.LIS, and CCC.LIS.

2. \$ FORTRAN XXX+YYY+ZZZ

Source files XXX.FOR, YYY.FOR, and ZZZ.FOR are concatenated and compiled as one file, producing an object file named XXX.OBJ, but no listing file. (A listing file named XXX.LIS would be produced in batch mode.)

3. \$ FORTRAN/OBJECT=SQUARE/NOLIST **(RET)**
_File: CIRCLE

The source file CIRCLE.FOR is compiled, producing an object file named SQUARE.OBJ, but no listing file.

4. \$ FORTRAN AAA+BBB,CCC/LIST

Two object files are produced: AAA.OBJ (comprising AAA.FOR and BBB.FOR) and CCC.OBJ (comprising CCC.FOR). One listing file is produced: CCC.LIS (comprising CCC.FOR).

5. \$ FORTRAN ABC+CIRC/NOBJECT+XYZ

When you include a qualifier in a list of files that are to be concatenated, the qualifier affects all files in the list. The command illustrated above completely suppresses the object file. That is, source files ABC.FOR, CIRC.FOR, and XYZ.FOR are concatenated and compiled, but no object file is produced.

3.2.3 Qualifiers to the FORTRAN Command

FORTRAN command qualifiers influence the way in which the compiler processes a file. In many cases, the simplest form of the FORTRAN command is sufficient. However, you can select appropriate optional qualifiers if special processing is required.

Table 3-1 lists the FORTRAN command qualifiers. Sections 3.2.3.1 through 3.2.3.18 describe each qualifier in detail.

You can override some qualifiers specified on the command line by using the OPTIONS statement. The qualifiers specified by the OPTIONS statement affect only the program unit where the statement occurs. Refer to Section 3.5.2 for more information.

Table 3-1: FORTRAN Command Qualifiers

Qualifier	Negative Form	Qualifier
/CHECK= { (NO)BOUNDS (NO)OVERFLOW (NO)UNDERFLOW ALL NONE }	/NOCHECK	/CHECK=(NOBOUNDS, OVERFLOW)
/CONTINUATIONS=n	None	/CONTINUATIONS=19
/CROSS_REFERENCE	/NOCROSS_REFERENCE	/NOCROSS_REFERENCE
/DEBUG= { (NO)SYMBOLS (NO)TRACEBACK ALL NONE }	/NODEBUG	/DEBUG=(NOSYMBOLS, TRACEBACK)
/D_LINES	/NOD_LINES	/NOD_LINES
/DML	None	None
/EXTEND_SOURCE	/NOEXTEND_SOURCE	/NOEXTEND_SOURCE
/F77	/NOF77	/F77
/G_FLOATING	/NOG_FLOATING	/NOG_FLOATING
/I4	/NOI4	/I4
/LIBRARY	None	Not applicable
/LIST[=file-spec] /LIST (batch)	/NOLIST	/NOLIST (interactive)
/MACHINE_CODE	/NOMACHINE_CODE	/NOMACHINE_CODE
/OBJECT[=file-spec]	/NOOBJECT	/OBJECT
/OPTIMIZE	/NOOPTIMIZE	/OPTIMIZE
/SHOW= { (NO)DICTIONARY (NO)INCLUDE (NO)MAP (NO)PREPROCESSOR (NO)SINGLE ALL NONE }	/NOSHOW	/SHOW=(NODICTIONARY, NOINCLUDE,MAP, NOPREPROCESSOR, SINGLE)
/STANDARD= { (NO)SOURCE_ FORM (NO)SYNTAX ALL NONE }	/NOSTANDARD	/NOSTANDARD
/WARNINGS= { (NO)DECLARATIONS (NO)GENERAL ALL NONE }	/NOWARNINGS	/WARNINGS=(NODECLARA- TIONS, GENERAL)

3.2.3.1 /CHECK Qualifier

The /CHECK qualifier produces run-time checks for the conditions indicated.

The qualifier has the form:

$$\text{/CHECK} = \left\{ \begin{array}{l} \text{ALL} \\ \text{[NO]BOUNDS} \\ \text{[NO]OVERFLOW} \\ \text{[NO]UNDERFLOW} \\ \text{NONE} \end{array} \right\}$$

where:

BOUNDS

Specifies that array and substring references are checked by the system to ensure that they are within the address boundaries specified in the array or character variable declaration.

For array bounds, only the address reference is checked; that is, the system only checks to determine whether you are in the same array; it does not check each individual dimension. Also, array bounds checking is not performed for arrays that are dummy arguments in which the last dimension bound is specified as * or both upper and lower dimensions are 1. For example:

```
DIMENSION B(0:10,0:*)
```

or

```
DIMENSION A(1)
```

OVERFLOW

Specifies that BYTE, INTEGER*2, and INTEGER*4 calculations are checked for arithmetic overflow. Real and complex calculations are always checked for overflow and are not affected by /NOCHECK. Integer exponentiation is performed by a routine in the mathematical library. The routine in the mathematical library always checks for overflow, even if /CHECK=NOOVERFLOW is specified.

UNDERFLOW

Specifies that real and complex calculations are checked for floating underflow. Refer to the *VAX FORTRAN User's Guide* for information about floating underflow.

ALL

Specifies that OVERFLOW, BOUNDS, and UNDERFLOW checks are performed.

NONE

Specifies that no checks are performed.

The default is /CHECK=OVERFLOW. Note that /CHECK is the equivalent of /CHECK=ALL, and /NOCHECK is the equivalent of /CHECK=NONE.

3.2.3.2 /CONTINUATIONS Qualifier

The /CONTINUATIONS qualifier specifies the number of continuation lines allowed in a source program statement.

The qualifier has the form:

```
/CONTINUATIONS=n
```

where:

n

is an integer from 0 to 99.

If you omit the /CONTINUATIONS qualifier, the default value is 19.

Because the compiler has to assume maximum-length continuation lines (66 or 126 characters) when allowing space for continuation line sequences, the actual number of continuation lines allowed in any given statement usually exceeds the limit specified by the /CONTINUATIONS qualifier.

NOTE

A common problem is an attempt to use the character zero (0) as a continuation character. This is not allowed. A line with a “continuation” character of 0 is treated as an initial line; it does not indicate that the /CONTINUATIONS value needs to be increased.

3.2.3.3 /CROSS__REFERENCE Qualifier

The /CROSS__REFERENCE qualifier specifies that the storage map section of the listing file is to include information about the use of symbolic names. The cross-reference contains the numbers of the lines in which the symbols are defined and referenced.

The qualifier has the form:

```
/CROSS__REFERENCE
```

The /CROSS__REFERENCE qualifier is ignored if the listing file is not being generated.

The default is /NOCROSS__REFERENCE.

See Section 3.7.3 for a description of the listing format used when /CROSS__REFERENCE is specified.

3.2.3.4 /DEBUG Qualifier

The /DEBUG qualifier specifies that the compiler is to provide information for use by the VAX Symbolic Debugger and the run-time error traceback mechanism.

The qualifier has the form:

$$\text{/DEBUG} = \left. \begin{array}{l} \text{ALL} \\ \text{[NO]SYMBOLS} \\ \text{[NO]TRACEBACK} \\ \text{NONE} \end{array} \right\}$$

where:

SYMBOLS

Specifies that the compiler is to provide the debugger with local symbol definitions for user-defined variables, arrays (including dimension information), structures, and labels of executable statements.

TRACEBACK

Specifies that the compiler is to provide an address correlation table so that the debugger and the run-time error traceback mechanism can translate virtual addresses into source program routine names and compiler-generated line numbers.

ALL

Specifies that the compiler is to provide both local symbol definitions and an address correlation table.

NONE

Specifies that the compiler is to provide no debugging information.

If you do not specify the /DEBUG qualifier, the default is /DEBUG=TRACEBACK. Note that /DEBUG is the equivalent of /DEBUG=ALL, and /NODEBUG is the equivalent of /DEBUG=NONE.

NOTE

The use of /NOOPTIMIZE is strongly recommended when the /DEBUG qualifier is used. Optimizations performed by the compiler can cause several different kinds of unexpected behavior when using VAX DEBUG. See Chapter 1 of the *VAX FORTRAN User's Guide* for more information on this subject.

For more information on debugging and traceback, see Section 4.3 and Chapter 17.

3.2.3.5 /DLINES Qualifier

The /D__LINES qualifier specifies that lines with a D in column 1 are to be compiled and are not to be treated as comment lines.

The qualifier has the form:

```
/D__LINES
```

The default is /NOD__LINES, which means that lines with a D in column 1 are treated as comments.

3.2.3.6 /DML Qualifier

The /DML qualifier specifies that the FORTRAN Data Manipulation Language (DML) preprocessor is to be invoked before the compiler. The preprocessor produces an intermediate file of FORTRAN source code in which FORTRAN DML commands are expanded into FORTRAN statements. The compiler is then automatically invoked to compile this intermediate file.

The qualifier has the form:

```
/DML
```

You use the /SHOW=PREPROCESSOR qualifier in conjunction with the /DML qualifier to cause the preprocessor-generated source code to be included in the listing file. For more information on the DML preprocessor, refer to the *VAX DBMS FDML Reference Manual*.

NOTE

Because the intermediate file is deleted by the FORTRAN DML preprocessor immediately after compilation is complete, the debugger cannot access the source program when the /DML qualifier is used.

3.2.3.7 /EXTEND__SOURCE Qualifier

The /EXTEND__SOURCE qualifier specifies that the compiler is to extend the range of FORTRAN source text from columns 1 through 72 to columns 1 through 132.

The qualifier has the form:

```
/EXTEND__SOURCE
```

This qualifier can also be specified on the OPTIONS statement. The default in either case is /NOEXTEND__SOURCE.

If a source line is longer than 132 characters, a fatal read error is signaled and the compilation is immediately terminated.

3.2.3.8 /F77 Qualifier

The /F77 qualifier specifies that FORTRAN-77 interpretation rules are used for those statements that have a meaning incompatible with FORTRAN-66. See Appendix A in the *VAX FORTRAN User's Guide* for a discussion of these incompatibilities.

The qualifier has the form:

`/F77`

The default is /F77. If you specify /NOF77, the compiler selects FORTRAN-66 interpretations in cases of incompatibility.

3.2.3.9 /G__FLOATING Qualifier

The /G__FLOATING qualifier controls how the compiler implements REAL*8, COMPLEX*16, DOUBLE PRECISION, and DOUBLE COMPLEX quantities.

The qualifier has the form:

`/G__FLOATING`

/NOG__FLOATING, the default, causes the compiler to implement double-precision quantities using the VAX D__floating data type. /G__FLOATING causes the compiler to implement such quantities using the VAX G__floating data type.

If your program requires the G__floating form of double precision for its correct operation (that is, it uses a range larger than 10^{*38}), you should use the /G__FLOATING qualifier in an OPTIONS statement in your source program. The implementation of REAL*8 in VAX FORTRAN is further discussed in Section 6.1.2.

Note that routines between which double-precision quantities are passed should not mix the D__floating and G__floating data types.

CAUTION

VAX/VMS systems support both D__floating and G__floating implementations of REAL*8. On different systems, however, the performance of a program can vary widely depending on whether your program is compiled with G__floating or D__floating. The disparity exists when a particular system supports one floating type in hardware and the other in software. Thus, if you wish to optimize performance and if range and accuracy constraints do not prescribe one of the two options, you must ensure that the most efficient option is in effect during the compilation process.

You can select G__floating or D__floating by means of an OPTIONS statement in your source program or by means of qualifiers on the FORTRAN command.

For more information on floating-point data types, see Sections 6.1.2, C.4.2, and C.4.3.

3.2.3.10 /I4 Qualifier

The /I4 qualifier controls how the compiler interprets INTEGER and LOGICAL declarations that do not have a specified length.

The qualifier has the form:

```
/I4
```

The default is /I4, which causes the compiler to interpret INTEGER and LOGICAL declarations as INTEGER*4 and LOGICAL*4. If you specify /NOI4, the compiler interprets them as INTEGER*2 and LOGICAL*2.

3.2.3.11 /LIBRARY Qualifier

The /LIBRARY qualifier specifies that a file is a text library file.

The qualifier has the form:

```
text-library-file/LIBRARY
```

The /LIBRARY qualifier can be specified on one or more text library files in a list of files concatenated by plus signs. At least one of the files in the list must be a nonlibrary file. The default file type is TLB.

The use of text libraries is discussed at length in Section 3.3.

3.2.3.12 /LIST Qualifier

The /LIST qualifier specifies that a source listing file is to be produced.

The qualifier has the form:

```
/LIST[=file-spec]
```

You can include a file specification for the listing file. If you do not, it defaults to the name of the first source file and to a file type of LIS.

In interactive mode, the compiler does not produce a listing file unless you include the /LIST qualifier. In batch mode, the compiler produces a listing file by default. In either case, the listing file is not automatically printed; you must use the PRINT command to obtain a line printer copy of the listing file.

See Section 3.7.1 for a discussion on the format of listing files.

3.2.3.13 /MACHINE_CODE Qualifier

The /MACHINE_CODE qualifier specifies that the listing file is to include a symbolic representation of the object code generated by the compiler. Generated code and data is represented in a form similar to a VAX MACRO assembly listing. Do not attempt to assemble this listing file; several items included in the listing file are not supported by VAX MACRO assembler.

The qualifier has the form:

```
/MACHINE_CODE
```

This qualifier is ignored if no listing file is being generated. The default is /NOMACHINE_CODE.

See Section 3.7.2 for a description of the format of a machine code listing.

3.2.3.14 /OBJECT Qualifier

The /OBJECT qualifier specifies the name of the object file.

The qualifier has the form:

```
/OBJECT[=file-spec]
```

The default is /OBJECT. The negative form, /NOOBJECT, can be used to suppress object code (for example, when you want to test only for compilation errors in the source program).

If you omit the file specification, the object file defaults to the name of the first source file and to a file type of OBJ.

3.2.3.15 /OPTIMIZE Qualifier

The /OPTIMIZE qualifier specifies that the compiler is to produce optimized code.

The qualifier has the form:

```
/OPTIMIZE
```

The default is /OPTIMIZE. The negative form /NOOPTIMIZE should be used during a debugging session to ensure that the debugger has sufficient information to locate errors in the source program. (See the *VAX FORTRAN User's Guide* for information on optimizations performed by the VAX FORTRAN compiler.)

3.2.3.16 /SHOW Qualifier

The /SHOW qualifier controls whether optionally listed source lines (that is, text module source lines and preprocessor generated source lines) and a symbol map are to appear in the source listing.

The /LIST qualifier must be specified in order for the /SHOW qualifier to take effect.

The qualifier has the form:

$$\text{/SHOW} = \left\{ \begin{array}{l} \text{ALL} \\ \text{[NO]DICTIONARY} \\ \text{[NO]INCLUDE} \\ \text{[NO]MAP} \\ \text{[NO]PREPROCESSOR} \\ \text{[NO]SINGLE} \\ \text{NONE} \end{array} \right\}$$

where:

ALL

specifies that all optionally listed source lines are to be included in the listing file.

INCLUDE

specifies that the source lines from any file specified by INCLUDE statements are to be included in the source listing.

DICTIONARY

specifies that FORTRAN source representations of any CDD records referenced by DICTIONARY statements are to be included in the listing file.

MAP

specifies that the symbol map is to be included in the listing file. If the /CROSS—REFERENCE qualifier is specified, MAP is ignored.

PREPROCESSOR

specifies that preprocessor-generated source lines are to be included in the listing file. The negative form, NOPREPROCESSOR, specifies that the source lines are to be excluded from the source listing.

NONE

specifies that no optionally listed source lines are to be included in the listing file.

SINGLE

specifies that names only referenced once (that is, those names that appear only in declarations) are to be included with multiply-referenced names in cross-reference listings. NOSINGLE specifies that names only referenced once are to be suppressed. This is useful for cross-reference listings of small programs that specify INCLUDE declarations but use only a small number of the names declared.

The /SHOW qualifier defaults are NOPREPROCESSOR, NOINCLUDE, NODICTIONARY, MAP, SINGLE.

Specifying the qualifier /SHOW without any arguments is equivalent to specifying /SHOW=ALL; specifying /NOSHOW without any arguments is equivalent to specifying /SHOW=NONE.

3.2.3.17 /STANDARD Qualifier

The /STANDARD qualifier specifies that the compiler is to generate informational diagnostics for non-semantic VAX extensions to FORTRAN-77.

The qualifier has the form:

$$\text{/STANDARD} = \left\{ \begin{array}{l} \text{ALL} \\ \text{[NO]SYNTAX} \\ \text{[NO]SOURCE_FORM} \\ \text{NONE} \end{array} \right\}$$

where:

SYNTAX

specifies that an informational message is to be issued for syntax extensions to the current ANSI standard.

SOURCE_FORM

specifies that an informational message is to be issued for statements that use tab formatting or contain lowercase characters.

ALL

specifies that informational messages are to be issued for both syntax and source form extensions to the current ANSI standard.

NONE

specifies that no informational messages are to be issued for extensions to the current ANSI standard.

The default is /NOSTANDARD, which is equivalent to /STANDARD=NONE.

If you have specified the /NOWARNINGS qualifier, the /STANDARD qualifier is ignored. Specifying /STANDARD with no arguments is equivalent to specifying /STANDARD=(SYNTAX, NOSOURCE_FORM).

The compiler does not diagnose semantic extensions if /STANDARD=ALL is specified. Semantic extensions are standard conforming statements that become nonstandard because of the way in which they are used.

3.2.3.18 /WARNINGS Qualifier

The /WARNINGS qualifier specifies that the compiler is to generate informational (I) and warning (W) diagnostic messages in response to informational and warning-level errors.

The qualifier has the form:

$$\text{/WARNINGS} = \left\{ \begin{array}{l} \text{ALL} \\ \text{[NO]DECLARATIONS} \\ \text{[NO]GENERAL} \\ \text{NONE} \end{array} \right\}$$

where:

GENERAL

causes the compiler to generate informational and warning diagnostic messages. An informational message indicates that a correct VAX FORTRAN statement may have unexpected results or contains nonstandard syntax or source form. A warning message indicates that the compiler has detected acceptable, but nonstandard, syntax or has performed some corrective action; in either case, unexpected results may occur. To suppress I and W diagnostic messages, specify the negative form of this qualifier (NOGENERAL). The default is GENERAL.

DECLARATIONS

causes the compiler to print warnings for any undeclared data item used in the program. DECLARATIONS acts as an external IMPLICIT NONE declaration. The default is NODECLARATIONS.

ALL

causes the compiler to print all informational and warning messages, including warning messages for any undeclared data items.

NONE

suppresses all informational and warning messages.

Appendix B discusses compiler diagnostic messages.

3.3 Using Text Libraries

A text library contains modules of source text that you can incorporate in a program by using the INCLUDE statement. Modules within a text library are like ordinary text files, but they differ in the following ways: they contain a unique name, called the module name, that is used to access them, and several can be contained within the same library file. Modules in text libraries can contain any kind of text; this section only discusses their use when FORTRAN language source is used.

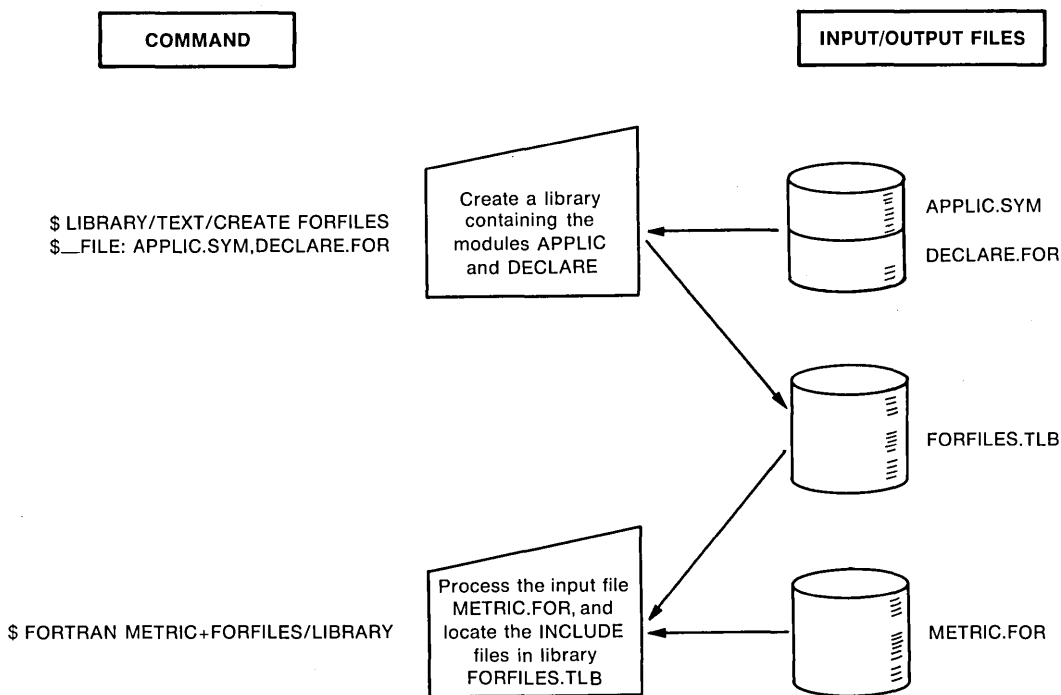
To create and modify modules in text libraries, you use the VAX/VMS LIBRARY command. Text libraries have a default file type of TLB.

To access a source module in a text library, you do one of the following:

- Specify only the name of the module in an INCLUDE statement in your FORTRAN source program.
- Specify the name of both the library and module in an INCLUDE statement in your FORTRAN source program.
- Specify the name of the library using the /LIBRARY qualifier in the FORTRAN command that you use to compile the source program; or define a default library.

For information on how to use INCLUDE statements and the /LIBRARY qualifier, see Sections 3.5.2 and 3.3.3, respectively.

Figure 3-1 illustrates the creation of a text library and its use in compiling FORTRAN programs.



ZK-792-82

Figure 3-1: Creating and Using a Text Library

3.3.1 Using the LIBRARY Commands

Table 3-2 summarizes the commands that create libraries and provide maintenance functions. For a complete list of the qualifiers for the LIBRARY command and a description of other DIGITAL Command Language (DCL) commands listed in Table 3-2, see the *Guide to Using DCL and Command Procedures on VAX/VMS*.

Table 3-2: Commands to Control Library Files

Function	Command Syntax ¹
Create a library	\$ LIBRARY/TEXT/CREATE library-name file-spec,...
Add one or more modules to a library	\$ LIBRARY/TEXT/INSERT library-name file-spec,...
Replace one or more modules in a library	\$ LIBRARY/TEXT/REPLACE ² library-name - _ \$ file-spec,...
Specify the names of modules to be added to a library	\$ LIBRARY/TEXT/INSERT library-name - _ \$ file-spec/MODULE=module-name
Delete one or more modules from a library	\$ LIBRARY/TEXT/DELETE=(module-name,...)- _ \$ library-name
Copy a module from a library into another file	\$ LIBRARY/TEXT/EXTRACT=module-name- _ \$ /OUTPUT=file-spec library-name
List the modules in a library	\$ LIBRARY/TEXT/LIST=file-spec library-name

¹ The LIBRARY command qualifier/TEXT indicates a text module library. By default, the LIBRARY command assumes an object module library.

² REPLACE is the default function of the LIBRARY command if no other action qualifiers are specified. If no module exists with the given name, /REPLACE is effectively /INSERT.

3.3.2 Naming Text Modules

When the LIBRARY command adds a module to a library, it uses by default the file name of the input file as the name of the module. In the example in Figure 3-1, the LIBRARY command adds the contents of the files APPLIC.SYM and DECLARE.FOR to the library and names the modules APPLIC and DECLARE.

Alternatively, you can name a module in a library with the /MODULE qualifier. For example:

```
$ LIBRARY/TEXT/INSERT FORFILES DECLARE.FOR/MODULE=EXTERNAL_DECLARATIONS
```

The preceding command inserts the contents of the file DECLARE.FOR into the library FORFILES under the name EXTERNAL_DECLARATIONS. This module can be included in a FORTRAN source file during compilation with the statement:

```
INCLUDE 'FORFILES(EXTERNAL_DECLARATIONS)'
```


3.3.3 Specifying Library Files on the FORTRAN Command Line

The `/LIBRARY` qualifier is used on the FORTRAN command line to identify text libraries. If a source file that you are compiling includes a module from a text library, you concatenate the name of the text library to the name of the source file and append the `/LIBRARY` qualifier to the text library name. Concatenation is specified with a plus sign. For example:

```
$ FORTRAN APPLIC+DATAB/LIBRARY
```

Whenever an `INCLUDE` statement occurs in `APPLIC.FOR`, the compiler searches the library `DATAB.TLB` for the source text module identified in the `INCLUDE` statement and incorporates it into the compilation. See Section 3.5.1 for a description of the `INCLUDE` statement.

3.3.4 Search Order of Libraries

When more than one library is specified on a FORTRAN command, the FORTRAN compiler searches the libraries in the order specified, on the command line, each time it processes an `INCLUDE` statement that specifies a text module name. For example:

```
$ FORTRAN APPLIC+DATAB/LIBRARY+NAMES/LIBRARY+GLOBALSYMS/LIBRARY
```

When FORTRAN processes an `INCLUDE` statement in the source file `APPLIC.FOR`, it searches the libraries `DATAB.TLB`, `NAMES.TLB`, and `GLOBALSYMS.TLB`, in that order, for source text modules identified in the `INCLUDE` statement.

On a command that requests multiple compilations, a library must be specified for each compilation in which it is needed. For example:

```
$ FORTRAN METRIC+DATAB/LIBRARY, APPLIC+DATAB/LIBRARY
```

In this example, FORTRAN compiles `METRIC.FOR` and `APPLIC.FOR` separately and uses the library `DATAB.TLB` for each compilation.

After the compiler has searched all libraries specified in the command, it searches the user-supplied default library, if any, specified by the logical name `FORT$LIBRARY`, and then the system-supplied default library `SYS$LIBRARY:FORSYSDEF.TLB`.

3.3.4.1 User-Supplied Default Libraries

You can define one of your private text libraries as a default library for the FORTRAN compiler to search. The FORTRAN compiler searches the default library after it searches libraries specified in the FORTRAN command.

To define a default library, assign an equivalence for the logical name `FORT$LIBRARY`, as in the following example of the VAX/VMS `ASSIGN` command:

```
$ ASSIGN DBA0:[LIB]DATAB FORT$LIBRARY
```

While this assignment is in effect, the compiler automatically searches the library `DBA0:[LIB]DATAB.TLB` for any include modules that it cannot locate in libraries explicitly specified on the FORTRAN command, if any.

You can define the logical name `FORT$LIBRARY` in any logical name table. If the name is defined in more than one table, the FORTRAN compiler uses the equivalence for the first match it finds in the normal order of search (that is, the process, then group, then system table). Thus, if `FORT$LIBRARY` is defined in both the process and group logical name tables, the process logical name table assignment overrides the group logical name table assignment.

If `FORT$LIBRARY` is defined as a search list, the compiler opens the first text library specified in the list. If the include module is not found there, the search is terminated and an error message is issued.

3.3.4.2 System-Supplied Default Library

When the FORTRAN compiler cannot find the include modules in libraries specified on the FORTRAN command or in the default library defined by `FORT$LIBRARY`, FORTRAN searches the system-supplied library `SYS$LIBRARY:FORSYSDEF.TLB`.

`SYS$LIBRARY` identifies the device and directory containing system libraries and is normally defined by the system manager. `FORSYSDEF.TLB` is a library of include modules supplied by VAX FORTRAN. It contains local symbol definitions required for use with system services, and return status values from system services.

Refer to Appendix C in the *VAX FORTRAN User's Guide* for more information on the contents of `FORSYSDEF`.

3.4 Using the VAX Common Data Dictionary

The Common Data Dictionary (CDD) is an optional VAX software product available under a separate license. The CDD allows you to maintain a set of shareable data definitions (language-independent structure declarations) that are defined by a system manager or data administrator. See the *VAX Common Data Dictionary Utilities Reference Manual* and the *VAX CDD Data Definition Language Reference Manual* for detailed information about the CDD.

CDD data definitions are organized hierarchically in much the same way that files are organized in directories and subdirectories. For example, a dictionary for defining personnel data might have separate directories for each employee type. A directory for salesmen might have subdirectories that would include data definitions for records such as salary and commission history or personnel history.

Descriptions of data definitions are entered into the dictionary in a special-purpose language called CDDL (Common Data Dictionary Language). Then, the CDDL compiler converts the data descriptions to an internal form and inserts them into the CDD, thus making them independent of the language used to access them.

During the compilation of a FORTRAN program, CDD data definitions can be accessed by means of `DICTIONARY` statements. If the data attributes of the data definitions are consistent with FORTRAN requirements, the data definitions are included in the FORTRAN program. CDD data definitions, in the form of FORTRAN source code, appear in source program listings if you specify the `/SHOW=DICTIONARY` qualifier on the FORTRAN command or `/LIST` in the `DICTIONARY` statement.

The advantage in using the CDD, instead of FORTRAN source, for structure declarations is that CDD record declarations are language independent and can be used with several supported VAX languages.

The following examples illustrate how data definitions are written for the CDD. The first example is a structure declaration written in CDDL. The second example shows the same structure as it would appear in a FORTRAN listing.

- **CDDL Representation:**

```
PAYROLL_RECORD STRUCTURE,  
  SALESMAN STRUCTURE,  
    NAME                DATATYPE IS TEXT 30,  
    ADDRESS              DATATYPE IS TEXT 40,  
    SALESMAN_ID /       DATATYPE IS UNSIGNED NUMERIC 5,  
  END SALESMAN STRUCTURE,  
END PAYROLL_RECORD STRUCTURE.
```

- **FORTRAN Source Code Representation:**

```
STRUCTURE /PAYROLL_RECORD/  
  STRUCTURE SALESMAN  
    CHARACTER*30 NAME  
    CHARACTER*40 ADDRESS  
    STRUCTURE SALESMAN_ID  
      CHARACTER*3 %FILL  
    END STRUCTURE  
  END STRUCTURE  
END STRUCTURE
```

The CDD provides two utilities for creating and maintaining a dictionary:

- The Dictionary Management Utility (DMU)
- The Dictionary Verify/Fix Utility (CDDV)

The Dictionary Management Utility (DMU) is for creating and maintaining the CDD's directory hierarchy, history lists, and access control lists. The Dictionary Verify/Fix Utility (CDDV) is for repairing damaged dictionary files. See the *VAX Common Data Dictionary Utilities Reference Manual* for details.

3.4.1 Accessing the CDD from FORTRAN Programs

DMU commands create directories and define record paths. Once these paths are established, records can be extracted from the CDD by means of DICTONARY statements in VAX FORTRAN programs.

At compile time, the CDD record and its attributes are extracted from the designated CDD record node. Then, the compiler converts the extracted record into a FORTRAN structure declaration and includes it in the object module.

The DICTONARY statement incorporates VAX Common Data Dictionary data definitions into the current FORTRAN source file during compilation. It can occur anywhere in a FORTRAN source file that a specification statement (such as a STRUCTURE/END STRUCTURE block) is allowed. The format of the DICTONARY statement is described in Section 3.5.3.

A DICTONARY statement must appear as a statement by itself; it cannot be used within a FORTRAN structure declaration. For example, the following DICTONARY statement

```
INTEGER*4 PRICE
DICTIONARY 'ACCOUNTS'
```

would result in a declaration of the form

```
INTEGER*4 PRICE
STRUCTURE /ACCOUNTS/
  STRUCTURE NUMBER
    CHARACTER*3 LEDGER
    CHARACTER*5 SUBACCOUNT
  END STRUCTURE
  CHARACTER*12 DATE
  .
  .
  .
END STRUCTURE
```

When you extract a record definition from the CDD, you can choose to include this translated record in the program's listing by using the /LIST in the DICTONARY statement or the /SHOW=DICTIONARY qualifier in the FORTRAN command line.

CDD data definitions can contain explanatory text in the CDDL DESCRIPTION IS clause. If you specify /SHOW=DICTIONARY on the FORTRAN command (or /LIST in the DICTONARY statement), this text is included in the FORTRAN listing as comments. The programmer may use these comments to indicate the data type of each structure and member. The punctuation for CDDL comments is the same as for other FORTRAN programs.

Because the DICTONARY statement generally contains only structure declaration blocks (see Section 8.15), you will usually also need to include one or more RECORD statements (see Section 8.13) in your program to make use of these structures.

3.4.2 Creating CDD Structure Declarations

CDD source files must be written in the Common Data Dictionary Language (CDDL). You enter them using the EDT editor, just as you would any other file. After you have created a CDD source file, you can then invoke the CDD compiler to insert your record definitions into the CDD. See the *VAX CDD Data Definition Language Reference Manual* for detailed information about the CDDL language and compiler.

3.4.3 FORTRAN and CDDL Data Types

The CDD supports some data types that are not native to FORTRAN. If a data definition contains an unsupported data type, FORTRAN makes the unsupported data type accessible by declaring it as an inner STRUCTURE containing a single CHARACTER %FILL field of an appropriate length. FORTRAN does not attempt to approximate a data type that is not supported by FORTRAN. For example, an UNSIGNED LONG number is declared:

```
STRUCTURE /whatever/
  .
  .
  .
  STRUCTURE name
    CHARACTER*4 %FILL
  END STRUCTURE
END STRUCTURE
```

and not INTEGER*4, which would result in signed operations if the field was used in an arithmetic expression.

The following table summarizes the CDDL data types and corresponding FORTRAN data types. For further information on CDDL data types see the *Common Data Dictionary Data Definition Language Reference Manual*.

CDDL Data Type	FORTRAN Data Type
DATE	STRUCTURE (length 8)
DATE AND TIME	STRUCTURE (length n)
VIRTUAL	ignored
BIT m ALIGNED	STRUCTURE (length n+7/8)
BIT m	STRUCTURE (length n+7/8)
UNSPECIFIED	STRUCTURE (length n)
TEXT	CHARACTER*n
VARYING TEXT	STRUCTURE (length n)
VARYING STRING	STRUCTURE (length n)
D__FLOATING	REAL*8 (/NOG__FLOAT only)
D__FLOATING COMPLEX	COMPLEX*16 (/NOG__FLOAT only)
F__FLOATING	REAL*4
F__FLOATING COMPLEX	COMPLEX*8

CDDL Data Type

G__FLOATING
G__FLOATING COMPLEX
H__FLOATING
H__FLOATING COMPLEX
SIGNED BYTE
UNSIGNED BYTE
SIGNED WORD
UNSIGNED WORD
SIGNED LONGWORD
UNSIGNED LONGWORD
SIGNED QUADWORD
UNSIGNED QUADWORD
SIGNED OCTAWORD
UNSIGNED OCTAWORD
PACKED NUMERIC
SIGNED NUMERIC
UNSIGNED NUMERIC
LEFT OVERPUNCHED
LEFT SEPARATE
RIGHT OVERPUNCHED
RIGHT SEPARATE

FORTRAN Data Type

REAL*8 (/G__FLOAT only)
COMPLEX*16 (/G__FLOAT only)
REAL*16
STRUCTURE (length 32)
LOGICAL*1
STRUCTURE (length 1)
INTEGER*2
STRUCTURE (length 2)
INTEGER*4
STRUCTURE (length 4)
STRUCTURE (length 8)
STRUCTURE (length 8)
STRUCTURE (length 16)
STRUCTURE (length 16)
STRUCTURE (length n)
STRUCTURE (length n)
STRUCTURE (length n)
STRUCTURE (length n)
STRUCTURE (length n)
STRUCTURE (length n)

NOTE

D__floating and G__floating data types cannot be mixed in one subroutine; both types cannot be handled simultaneously. You can use both types, each in a separate subroutine, depending on the OPTIONS statement qualifier in effect for the individual subroutine. For a discussion of the handling of REAL*8 data types in VAX FORTRAN, see Section 6.2.1.2.

FORTRAN ignores CDD features that are not supported by FORTRAN, but issues error messages when the features conflict with FORTRAN.

3.5 Compilation Control Statements

In addition to qualifiers on the FORTRAN command, several statements used in the body of a FORTRAN program also influence the compilation process.

- The INCLUDE statement incorporates external source code into your programs during the compilation process.
- The OPTIONS statement establishes compiler qualifiers that would otherwise be specified on the FORTRAN command. If the same qualifier is specified on both the OPTIONS statement and the FORTRAN command, the OPTIONS statement version overrides if a conflict occurs.
- The DICTIONARY statement extracts records from the CDD (Common Data Dictionary) and converts them into VAX FORTRAN records for use in the program.

3.5.1 INCLUDE Statement

The INCLUDE statement specifies that the contents of a file or a text library module are to be incorporated in the FORTRAN compilation directly following the INCLUDE statement. The INCLUDE statement has no effect on program execution. It simply directs the compiler to read FORTRAN statements from a different file or a text library.

The INCLUDE statement has the form:

```
INCLUDE  { '[file-spec] (module-name)/[NO]LIST' }  
         { 'file-spec/[NO]LIST' }
```

where:

file-spec

A character string that specifies either (1) a file to be included in the compilation or (2) a text library containing a module to be included in the compilation. This file specification must be acceptable to the operating system. (See Section 1.5.1 for the form of a file specification.)

module-name

The name of a text module, located in a text library, that is to be included in the source file. The name of the module must be enclosed in parentheses. It can be up to 31 characters long and can contain any alphanumeric character and the special characters dollar sign (\$) and underscore (_).

/[NO]LIST

The /LIST qualifier indicates that the statements in the specified file or module are to be listed in the compilation source listing. A number indicating the depth of nesting of include files precedes each statement listed. The /NOLIST qualifier indicates that the included statements are not to be listed in the compilation source listing. The default is /NOLIST.

When the compiler encounters an `INCLUDE` statement, it stops reading statements from the current file and reads the statements in the included file or module. When it reaches the end of the included file or module, the compiler resumes compilation with the next statement after the `INCLUDE` statement.

An `INCLUDE` statement can be contained in an included file or module.

An included file or module cannot begin with a continuation line. Each FORTRAN statement must be completely contained within a single file or module.

The `INCLUDE` statement can appear anywhere within a program unit, as shown in Figure 5-1. Any FORTRAN statement can appear in an included file or module. However, the included statements, when combined with the other statements in the compilation, must satisfy the statement-ordering restrictions described in Section 5.2.2.1.

In the following example, the file `COMMON.FOR` defines the size of the blank common block and the size of the arrays `X`, `Y`, and `Z`.

Main Program File

```
      INCLUDE 'COMMON.FOR'
      DIMENSION Z(M)
      CALL CUBE
      DO 5, I=1,M
5     Z(I) = X(I)+SQRT(Y(I))
      *
      *
      *
      END

      SUBROUTINE CUBE
      INCLUDE 'COMMON.FOR'
      DO 10, I=1,M
10    X(I) = Y(I)*3
      RETURN
      END
```

File COMMON.FOR

```
PARAMETER (M=100)
COMMON X(M),Y(M)
```


3.5.2 OPTIONS Statement

The OPTIONS statement can be used to override or confirm the FORTRAN command qualifiers in effect in a program unit. It has the form:

```
OPTIONS qualifier[,qualifier...]
```

where:

qualifier

is one of the following:

```
/[NO]G__FLOATING
```

```
/[NO]I4
```

```
/[NO]F77
```

```
/CHECK = { ALL  
          [NO]OVERFLOW  
          [NO]BOUNDS  
          [NO]UNDERFLOW  
          NONE }
```

```
/NOCHECK
```

```
/[NO]EXTEND__SOURCE
```

The qualifiers have the same syntax and abbreviations as the FORTRAN command line qualifiers. Refer to 3.2.3 for a detailed explanation of the specific qualifiers.

The OPTIONS statement must be the first statement in a program unit, preceding the PROGRAM, SUBROUTINE, FUNCTION, and BLOCK DATA statements.

The OPTIONS qualifiers take precedence over qualifiers defined on the FORTRAN command line. However, the qualifiers remain in effect only until the end of the program unit in which they are defined. Thus, an OPTIONS statement must appear in each program unit in which you wish to override the command line qualifiers. For example:

```
OPTIONS /CHECK/EXTEND__SOURCE  
...  
END  
OPTIONS /G__FLOATING  
...
```

The first OPTIONS statement in the preceding example specifies that the program unit immediately following is to be compiled with full checking and extend-source options, regardless of the /CHECK and /EXTEND__SOURCE specifications on the FORTRAN command line. The next OPTIONS statement specifies that the program unit following it is to be compiled with the G__floating option. The check and extend-source options do not remain in effect across program unit boundaries.

3.5.3 DICTIONARY Statement

The DICTIONARY statement incorporates VAX Common Data Dictionary data definitions into the current FORTRAN source file during compilation. It can occur anywhere in a FORTRAN source file that a specification statement (such as a STRUCTURE/END STRUCTURE block) is allowed.

The format of the DICTIONARY statement is:

```
DICTIONARY 'cdd-path [/([NO)]LIST]'
```

where:

cdd-path

The cdd-path is interpreted as the full or relative pathname of a CDD object. The resulting pathname must conform to the rules for forming VAX CDD pathnames.

/([NO)]LIST

The /LIST qualifier controls whether the source code representation of the resulting structure declaration is to be listed in the compilation source listing. The default is /NOLIST.

There are two types of CDD pathname: full and relative. A full pathname begins with CDD\$TOP and specifies the given names of all its descendants; it is a complete path to the record definition. Descendant names are separated from each other by a period.

A relative pathname begins with any generation name other than CDD\$TOP and specifies the given names of the descendants after that point. A relative path may be accomplished by establishing a default directory with a logical name. For example:

```
$ DEFINE CDD$DEFAULT CDD$TOP.FOR
```

This logical name definition specifies the beginning of the CDD pathname; thus, a relative pathname specifies the remainder of the path to the record definition. Note also that a CDD pathname beginning with CDD\$TOP overrides the default CDD pathname. Refer to the *VAX Common Data Dictionary Utilities Manual* for further details.

For example, if you have a record with the following pathname:

```
CDD$TOP.SALES.JONES.SALARY
```

and you have defined CDD\$DEFAULT to be CDD\$TOP.SALES.JONES, then you may specify a relative pathname as

```
DICTIONARY 'SALARY'
```

or an absolute pathname as

```
DICTIONARY 'CDD$TOP.SALES.JONES.SALARY'
```

3.6 Compiler Diagnostic Messages and Error Conditions

One of the functions of the FORTRAN compiler is to identify syntax errors and violations of language rules in the source program. If the compiler locates any errors, it writes messages to your default output device; thus, if you enter the FORTRAN command interactively, the messages are displayed on your terminal. If the FORTRAN command is executed in a batch job, the messages appear in the batch job log file.

When it appears on the terminal, a message from the compiler has the following format:

```
%FORT-s-ident, message-text  
          [text-in-error] in module module-name at line n
```

Diagnostic messages usually provide enough information for you to determine the cause of an error and correct it.

Each compilation with diagnostic messages terminates with a summary that indicates the combined number of error, warning, and informational messages generated by the compiler. The diagnostic summary has the following form:

```
%FORT-s-ident, source-file-spec completed with n diagnostics
```

If the compiler creates a listing file, it also writes the messages to the listing. Messages typically follow the statement that caused the error.

Additional information about diagnostic messages, including descriptions of the individual messages, is contained in Appendix E.

3.7 Compiler Output Listing Format

A compiler output listing produced by a FORTRAN command with the /LIST qualifier consists of the following sections:

- A source code section
- A machine code section (optional)
- A storage map section (cross-reference, optional)
- A compilation summary

Sections 3.7.1 through 3.7.4 describe the compiler listing sections in detail.

3.7.1 Source Code Section

The source code section of a compiler output listing displays the source program as it appears in the input file, with the addition of sequential line numbers generated by the compiler. Figure 3-2 shows a sample of a source code section of a compiler output listing.

```

0001          SUBROUTINE RELAX2(EPS)
0002
0003          PARAMETER (M=40, N=60)
0004          DIMENSION X(0:M,0:N)
0005          COMMON X
0006
0007          LOGICAL DONE
0008
0009          1      DONE = .TRUE.
0010
0011          DO 10 J=1,N-1
0012          DO 10 I=1,M-1
0013              XNEW = (X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1))/4
0014              IF (ABS(XNEW-X(I,J)) .GT. EPS) DONE = .FALSE.
0015          10      X(I,J) = XNEW
0016
0017          IF (.NOT. DONE) GO TO 1
0018
0019          RETURN
0020          END

```

Figure 3-2: Sample Listing of Source Code

Compiler-generated line numbers appear in the left margin and are used with the %LINE prefix in debugger commands. If you create the source file with an editor that generates line numbers, those numbers also appear in the source listing. In this case, the editor-generated line numbers appear in the left margin, and the compiler-generated line numbers are shifted to the right. The %LINE specification still applies to the compiler-generated line numbers, not the editor-generated line numbers.

Compile-time error messages that contain line numbers refer to the editor-generated line numbers present in the source code listing; otherwise, they refer to compiler-generated line numbers. Run-time error messages that contain line numbers refer to the compiler-generated line numbers in the source code listing section. (See Appendix E for a summary of error messages.)

3.7.2 Machine Code Section

The machine code section of a compiler output listing provides a symbolic representation of the compiler-generated object code. The representation of the generated code and data is similar to that of a VAX MACRO assembly listing.

The machine code section is optional. To receive a listing file with a machine code section, you must specify:

```
$ FORTRAN/LIST/MACHINE_CODE
```

Figure 3-3 shows a sample of a machine code section of a compiler output listing.

```

        .TITLE  RELAX2
        .IDENT  01

0000      .PSECT  $BLANK
0000  X:

0000      .PSECT  $CODE

0000  RELAX2::                                ; 0001
0000      .WORD   ^M<IV,R2,R3,R4,R6,R7>
                                                ; 0009
0002      NOP
0003      NOP
0004      .1:
0004      MNEGL  #1, R0
                                                ; 0011
0007      MOVL   #1, R1
000A      NOP
000B      NOP
000C  L$1:
                                                ; 0012
000C      MOVL   #1, R2
000F      MULL3  #41, R1, R3
0013      MOVAF  X(R3), R4
001B      NOP
001C  L$2:
                                                ; 0013
001C      ADDF3  8(R4), (R4)+, R6
0021      ADDF2  -164(R4), R6
0026      ADDF2  164(R4), R6
002B      MULF2  #^X3F80, R6
                                                ; 0014
0032      SUBF3  (R4), R6, R7
0036      BICW2  #^X8000, R7
003B      CMPF  R7, @EPS(AP)
003F      BLEQ  L$3
0041      CLRL  R0
0043  L$3:
                                                ; 0015
0043      MOVL   R6, (R4)
0046      AOBLEQ #39, R2, L$2
004A      AOBLEQ #59, R1, L$1
                                                ; 0017
004E      BLBC  R0, .1
                                                ; 0019
0051      RET
        .END

```

Figure 3-3: Sample Listing of Machine Code

The following notes give a detailed explanation of how generated code and data are represented in machine code listings.

1. Machine instructions are represented by VAX MACRO mnemonics and syntax. Compiler-generated line numbers corresponding to generated code lines are listed at the right margin, preceding the machine code generated for the line.
2. The first line contains a .TITLE assembler directive, indicating the program unit to which the machine code corresponds.
 - For a main program, the title is as declared in a PROGRAM statement. If you did not specify a PROGRAM statement, the main program is titled filename\$MAIN, where filename is the name of the source file.
 - For a subprogram, the title is the name of the subroutine or function.
 - For a BLOCK DATA subprogram, the title is either the name declared in the BLOCK DATA statement, or filename\$DATA by default.
3. The lines following .TITLE provide information such as the contents of storage initialized for FORMAT statements, DATA statements, constants, and subprogram argument call lists.
4. The VAX general registers (0 through 12) are represented by R0 through R12. When register 12 is used as the argument pointer, it is represented by AP; the frame pointer (register 13) is FP; the stack pointer (register 14) is SP; and the program counter (register 15) is PC. Note that the relative PC for each instruction or data item is listed at the left margin, in hexadecimal.
5. Variables and arrays defined in the source program are shown as they were defined in the program. Offsets from variables and arrays are shown in decimal.
6. FORTRAN source labels referenced in the source program are shown with a period prefix (.). For example, if the source program refers to label 300, the label appears in the machine code listing as .300. Labels that appear in the source program, but are not referenced or are deleted during compiler optimization, are ignored. They do not appear in the machine code listing unless you specified /NOOPTIMIZE.
7. The compiler may generate labels for its own use. These labels appear as L\$n, where the value of n is unique for each such label in a program unit.
8. Integer constants are shown as signed integer values; real and complex constants are shown as unsigned hexadecimal values preceded by ^X.
9. Addresses are represented by the program section name plus the hexadecimal offset within that program section. Changes from one program section to another are indicated by PSECT lines.

3.7.3 Storage Map Section

The storage map section of the compiler output listing is printed after each program unit, or module. It summarizes information in the following categories:

- *Program sections*: The program section summary describes each program section (PSECT) generated by the compiler. The descriptions include:
 - PSECT number (used by most of the other summaries)
 - Name
 - Size in bytes
 - Attributes

PSECT usage and attributes are described in the *VAX FORTRAN User's Guide*.

- *Total memory allocated*: Following the program sections, the compiler prints the total memory allocated for all program sections compiled in the following form:

```
Total Space Allocated nnn
```

- *Entry points*: The entry point summary lists all entry points and their addresses. If the program unit is a function, the declared data type of the entry point is also included.
- *Statement functions*: The statement function summary lists the entry point address and data type of each statement function. If all of the references to a statement function generate inline code, the body of the statement function is not compiled, and a double asterisk (**) appears instead of an address.
- *Variables*: The variable summary lists all simple variables, with the data type and address of each. If the variable is removed as a result of optimization, a double asterisk (**) appears in place of the address.
- *Records*: The record summary lists all record variables. It shows the address, the structure that defines the fields of the individual records, and the total size of each record.
- *Arrays*: The array summary is similar to the variable summary. In addition to data type and address, the array summary gives the total size and dimensions of the array. If the array is an adjustable array or assumed-size array, its size is shown as double asterisks (**), and each adjustable dimension bound is shown as a single asterisk (*).
- *Record Arrays*: The record array summary is similar to the record summary. The record array summary gives the dimensions of the record array in addition to address, defining structure, and total size. If the record array is an adjustable array or assumed-size array, its size is shown as double asterisks (**), and each adjustable dimension bound is shown as a single asterisk (*).
- *Namelists*: The namelist summary lists names of namelists.

- *Labels*: The label summary lists all user-defined statement labels. FORMAT statement labels are suffixed with an apostrophe ('). If the label address field contains double asterisks (**), the label was not used or referred to by the compiled code.
- *Functions and subroutines*: The functions and subroutines summary lists all external routine references made by the source program. This summary does not include references to routines that are dummy arguments; the actual function or subroutine name is supplied by the calling program.

A heading for an information category is printed in the listing only when entries are generated for that category.

Cross-reference information is optional. It is supplied only when you specify:

```
$ FORTRAN/LIST/CROSS_REFERENCE
```

When you request cross-referencing, the compiler supplies information on the following entities:

- *Parameter constants*: The parameter constant summary lists all of the PARAMETER constants along with the data type of each.
- *Field scalars*: The field scalar summary lists all of the scalar fields declared within a structure block. It shows the starting offset within the structure for each scalar field, the name of the structure containing each scalar field, and the datatype and size (in bytes) of each scalar field.
- *Field arrays*: The field array summary lists all of the array fields declared within a structure block. It shows the starting offset within the structure for each array field; the name of the structure containing each array field; and the datatype, size (in bytes), and dimensions of each array field.

The compiler also supplies attributes and line number references if you request cross-referencing, the attributes indicate whether a variable or array appears in common and whether it appears in an EQUIVALENCE statement.

The compiler supplies the following reference information for each name:

- A source line number indicates where the name was referenced.
- An equal sign (=) next to a line number indicates that the value of a variable or array was modified at that line.
- A number sign (#) next to a line number indicates the line where the symbol was defined.
- An "A" next to a line number indicates an actual argument which may have been modified.
- A "D" next to a line number indicates that data initialization occurred at that point in the program.
- A number in parentheses (n) next to a line number indicates that the name appeared n times on that line.

Figure 3-4 shows an example of a storage map section with cross-reference information.

PROGRAM SECTIONS

Name	Bytes	Attributes
0 \$CODE	82	PIC CDN REL LCL SHR EXE RD NOWRT LONG
3 \$BLANK	10004	PIC OVR REL GBL SHR NOEXE RD WRT LONG
Total Space Allocated	10086	

ENTRY POINTS

Address	Type	Name	References
0-00000000		RELAX2	1#

VARIABLES

Address	Type	Name	Attributes	References
**	L*4	DONE		7 9= 14= 17
AP-00000004@	R*4	EPS		1 14
**	I*4	I		12= 13(4) 14 15
**	I*4	J		11= 13(4) 14 15
**	R*4	XNEW		13= 14 15

ARRAYS

Address	Type	Name	Attributes	Bytes	Dimensions	References
3-00000000	R*4	X	COMM	10004	(0:40, 0:60)	4 5
13(4)	14	15=				

PARAMETER CONSTANTS

Type	Name	References
I*4	M	3# 4 12
I*4	N	3# 4 11

LABELS

Address	Label	References
0-00000004	1	9# 17
**	10	11 12 15#

Figure 3-4: Sample Storage Map Section

As shown in Figure 3-4, a section size is printed as a number of bytes, expressed in decimal. A data address is specified as an offset from the start of a program section, expressed in hexadecimal. The symbol AP can appear instead of a program section. When it does, the

address refers to a dummy argument, expressed as the offset from the argument pointer (AP). Indirection is indicated by an at sign (@) following an address field. In this case, the address specified by the program section (or AP) plus the offset points to the address of the data, not to the data itself.

3.7.4 Compilation Summary Section

The final entries on the compiler listing are the compiler qualifiers and compiler statistics.

The first line of qualifiers in this section echoes the command line that you used to invoke the compiler. The next set of qualifiers shows which ones were in effect during the compilation. The compiler statistics are the machine resources used by the compiler.

If the /CROSS_REFERENCE qualifier is specified, an explanation of the reference flags is printed before the qualifier summary.

Figure 3-5 shows how compiler options and command qualifiers and compilation statistics appear on the listing.

```

+-----+
|                                     |
|           KEY TO REFERENCE FLAGS   |
|   =   - Value Modified              |
|   #   - Defining Reference          |
|   A   - Actual Argument, possibly  |
|   D   - Data Initialization         |
|   (n) - Number of occurrences on   |
|                                     |
+-----+

OPTIONS QUALIFIERS

/CHECK=(NOBOUNDS,OVERFLOW,NOUNDERFLOW)
/F77 /NOG_FLOATING /NOI4

COMMAND QUALIFIERS

FORTRAN /LISTING/MACHINE_CODE/CROSS_REFERENCE RELAX2

/CHECK=(NOBOUNDS,OVERFLOW,NOUNDERFLOW)
/DEBUG=(NOSYMBOLS,TRACEBACK)
/STANDARD=(NOSYNTAX,NOSOURCE_FORM)
/SHOW=(NOPREPROCESSOR,NOINCLUDE,MAP,NODICTIONARY,SINGLE)
/WARNINGS=(GENERAL,NODECLARATIONS)
/CONTINUATIONS=19 /CROSS_REFERENCE /NOD_LINES /NOEXTEND_SOURCE /F77
/NOG_FLOATING /I4 /MACHINE_CODE /OPTIMIZE

COMPILATION STATISTICS

Run Time:           1.17 seconds
Elapsed Time:      2.23 seconds
Page Faults:       138
Dynamic Memory:    326 pages

```

Figure 3-5: Sample Compilation Summary

Chapter 4

Linking and Running FORTRAN Programs

This chapter describes how to produce an executable image from a FORTRAN object file, how to execute the resulting image, and how to isolate run-time errors.

4.1 Linking FORTRAN Programs

This section describes how to use the linker and object module libraries to combine object modules into executable programs. It discusses:

- The functions performed by the linker
- The LINK command and its input and output files

The topics in this chapter are confined to areas of particular interest to FORTRAN programmers. For additional information on linker capabilities and detailed descriptions of LINK command qualifiers and options, see the *VAX/VMS Linker Reference Manual*.

4.1.1 Functions of the Linker

The primary functions of the linker are to allocate virtual memory within the executable image, to resolve symbolic references among modules being linked, to assign values to relocatable global symbols, and to perform relocation. The linker's end product is an executable image that you can run on a VAX/VMS system.

For any FORTRAN program unit, the object module generated by the compiler may contain calls and references to VAX FORTRAN run-time procedures, which the linker locates automatically in the default system object module libraries. The libraries are described in the *VAX/VMS Linker Reference Manual*.

4.1.2 The LINK Command

The LINK command initiates the linking of the object file. The command has the form:

```
$ LINK[/command-qualifiers] file-spec[/file-qualifiers]...
```

where:

/command-qualifiers

Specifies output file options.

file-spec

Specifies the input object file to be linked.

/file-qualifiers

Specifies input file options.

In interactive mode, you can issue the LINK command with no accompanying file specification. The system then requests the file specifications with the following prompt:

```
_File:
```

You can enter multiple file specifications by separating them with commas or plus signs. When used with the LINK command, the comma has the same effect as the plus sign; that is, a single executable image is created from the input files specified. If no output file is specified, the linker produces an executable image with the same name as that of the first object module and with a file type of EXE. Table 4-1 lists the linker qualifiers of particular interest to FORTRAN users. See the *VAX/VMS Linker Reference Manual* for details on the linker.

Table 4-1: LINK Command Qualifiers

Function	Qualifiers	Defaults
Request output file and define a file specification.	/EXECUTABLE[=file-spec] /SHAREABLE[=file-spec]	/EXECUTABLE=name.EXE, where name is the name of the first input file. /NOSHAREABLE
Request and specify the contents of a memory allocation listing.	/BRIEF /[NO]CROSS_REFERENCE /FULL /[NO]MAP	/NOCROSS_REFERENCE /NOMAP (interactive) /MAP=name.MAP (batch) where, name is the name of the first input
Specify the amount of debugging information.	/[NO]DEBUG /[NO]TRACEBACK	/NODEBUG /TRACEBACK

Table 4-1 (Cont.): LINK Command Qualifiers

Function	Qualifiers	Defaults
Indicate that input files are libraries and to specifically include certain modules.	/INCLUDE=(module-name...) /LIBRARY /SELECTIVE__SEARCH	Not applicable
Request or disable the searching of default user libraries and system libraries.	/[NO]SYSLIB /[NO]SYSSHR /[NO]USERLIBRARY[=table]	/SYSLIB /SYSSHR /USERLIBRARY=ALL
Indicate that an input file is a linker options file.	/OPTIONS	Not applicable

4.1.2.1 Linker Output File Qualifiers

You can include qualifiers in the LINK command to influence the output of the linker. You can also specify whether the debugging or the traceback facility is to be included.

The debugger and traceback qualifiers are:

```
[/NO]DEBUG
[/NO]TRACEBACK
```

The /DEBUG and /TRACEBACK qualifiers are described in Section 4.1.2.2.

Linker output consists of an image file and, optionally, a map file. The qualifiers that control image and map files are described under the headings that follow.

Image File Qualifiers

The image file qualifiers are /[NO]EXECUTABLE and /[NO]SHAREABLE. The use and effects of these two qualifiers are as follows:

- **/EXECUTABLE Qualifier.** If you do not specify an image file qualifier, the default is /EXECUTABLE, and the linker produces an executable image.

To suppress production of an image, specify /NOEXECUTABLE. For example, in the following command, the file CIRCLE.OBJ is linked, but no image is generated:

```
* LINK/NOEXECUTABLE CIRCLE
```

The /NOEXECUTABLE qualifier is useful if you want to verify the results of linking an object file without actually producing the image.

To designate a file specification for an executable image, use the /EXECUTABLE qualifier in the form:

```
/EXECUTABLE=file-spec
```

For example, in the following command, the file CIRCLE.OBJ is linked and the executable image generated by the linker is named TEST.EXE:

```
$ LINK/EXECUTABLE=TEST CIRCLE
```

- **/SHAREABLE Qualifier.** A shareable image is an image that has all of its internal references resolved, but it must be linked with one or more object modules to produce an executable image. A shareable image file, for example, can contain a library of routines or can be used by the system manager to create a global section for all users. To create a shareable image, specify the /SHAREABLE qualifier, as shown in the following example:

```
$ LINK/SHAREABLE CIRCLE
```

To include a shareable image as input to the linker, you can insert the shareable image into a shareable-image library and specify the library as input to the LINK command. By default, the linker automatically searches the system-supplied shareable-image library SYS\$LIBRARY:IMAGELIB.OLB after searching any libraries you specify on the LINK command line. You can also include a shareable image by using a linker options file. See the *VAX/VMS Linker Reference Manual* for more information.

If you specify (or default to) /NOSHAREABLE, the image produced cannot be linked with other images.

Map File Qualifiers

The map file qualifiers tell the linker whether a map file is to be generated and, if so, what information is to be included.

The map qualifiers are specified as follows:

```
/MAP[=file-spec] [ { /BRIEF } ] [ /FULL } ] [ /CROSS_REFERENCE ]
```

The linker uses the following map file defaults: in interactive mode, the default is to suppress the map; in batch mode, the default is to generate the map.

If you do not include a file specification with the /MAP qualifier, the map file has the name of the first input file and a file type of MAP. It is stored on the default device in the default directory.

The /BRIEF and /FULL qualifiers define the amount of information included in the map file. They function as follows:

- /BRIEF produces a summary of the image's characteristics and a list of contributing modules.
- /FULL produces a summary of the image's characteristics and a list of contributing modules (as produced by /BRIEF). It also produces a list, in symbol-name order, of global symbols and values (program, subroutine, and common block names, and names declared EXTERNAL) and a summary of characteristics of image sections in the linked image.

If neither `/BRIEF` nor `/FULL` is specified, the map file, by default, contains a summary of the image's characteristics, a list of contributing modules (as produced by `/BRIEF`), and a list of global symbols and values, in symbol-name order.

You can use the `/CROSS_REFERENCE` qualifier with either the default or `/FULL` map qualifiers to request cross-reference information for global symbols. This cross-reference information indicates the object modules that define and/or refer to global symbols encountered during linking. The default is `/NOCROSS_REFERENCE`.

4.1.2.2 /DEBUG and /TRACEBACK Qualifiers

The `/DEBUG` qualifier indicates that the debugger (see Chapter 17) is to be included in the executable image and that local symbol information contained in the object modules is to be included. The default is `/NODEBUG`.

When you use the `/TRACEBACK` qualifier, run-time error messages will be accompanied by a symbolic traceback that shows the sequence of calls that transferred control to the program unit in which the error occurred. If you specify `/NOTRACEBACK`, this information is not produced. The default is `/TRACEBACK`.

If you specify `/DEBUG`, the traceback capability is automatically included, and the `/TRACEBACK` qualifier is ignored. (See Section 4.3.1 for a sample traceback list.)

4.1.2.3 Linker Input File Qualifiers

Input file qualifiers affect the file specifications of input files. Input files can be object files, shareable files previously linked, or library files.

The qualifiers that control linker input files are the `/LIBRARY` qualifier and the `/INCLUDE` qualifier.

- The `/LIBRARY` qualifier has the form:

`/LIBRARY`

This qualifier specifies that the input file is an object-module or shareable-image library that is to be searched to resolve undefined symbols referenced in other input modules. The default file type is `OLB`.

- The `/INCLUDE` qualifier has the form:

`/INCLUDE=module-name(s)`

The qualifier specifies that the input file is an object-module or shareable-image library and that the modules named are the only modules in the library to be explicitly included as input. In the case of shareable-image libraries, the module is the shareable-image name.

At least one module name is required. To specify more than one, enclose the module names in parentheses and separate the names with commas.

The default file type is `OLB`. The `/LIBRARY` qualifier can also be used, with the same file specification, to indicate that the same library is to be searched for unresolved references.

4.1.3 Linker Messages

If the linker detects any errors while linking object modules, it displays messages about their cause and severity. If any errors or fatal conditions occur (severities E or F), the linker does not produce an image file.

Linker messages are descriptive, and you do not normally need additional information to determine the specific error. Some of the more common errors that occur during linking are as follows:

- An object module has compilation errors. This error occurs when you attempt to link a module that had warnings or errors during compilation. Although you can usually link compiled modules for which the compiler generated messages, you should verify that the modules will actually produce the output you expect.
- The modules that are being linked define more than one transfer address. The linker generates a warning if more than one main program has been defined. This can occur, for example, when an extra END statement exists in the program. The image file created by the linker in this case can be run; the entry point to which control is transferred is the first one that the linker found.
- A reference to a symbol name remains unresolved. This error occurs when you omit required module or library names from the LINK command and the linker cannot locate the definition for a specified global symbol reference.

If an error occurs when you link modules, you can often correct it simply by reentering the command string and specifying the correct routines or libraries.

4.2 Running FORTRAN Programs

This section describes the following considerations for executing FORTRAN programs on the VAX/VMS operating system:

- Using the RUN command to execute programs interactively
- Passing status values to the command interpreter

4.2.1 The RUN Command

The RUN command initiates execution of a program.

The command has the form:

```
$ RUN[/[NO]DEBUG] file-spec
```

You must specify the file name. If you omit optional elements of the file specification, the system automatically provides a default value. The default file type is EXE.

The /DEBUG qualifier allows you to use the debugger, even if you omitted this qualifier from the FORTRAN and LINK commands. Refer to Section 4.3 for details.

Before the image is activated, the system initializes to zero all variables and arrays that are not initialized by means of DATA statements. (Note: It is not considered a good programming practice to rely on this, however.)

4.2.2 System Processing at Image Exit

When the main program executes an END statement, or when any program unit in the program executes a STOP statement, the image is terminated. In the VAX/VMS operating system, the termination of an image, or image exit, causes the system to perform a variety of clean-up operations during which open files are closed, system resources are freed, and so on.

4.2.3 Interrupting a Program

When you execute the RUN command interactively, you cannot execute any other program images or DCL commands until the current image completes. However, if your program is not performing as expected—if, for instance, you have reason to believe it is in an endless loop—you can interrupt it. To do so, use the <CTRL/Y> key. (You may also use the <CTRL/C> key, unless your program takes specific action in response to <CTRL/C>.) For example:

```
$ RUN APPLIC
^Y
$
```

This command interrupts the program APPLIC. After you have interrupted a program, you can terminate it by entering a DCL command that causes another image to be executed or by entering the DCL commands EXIT or STOP.

Following a <CTRL/Y> interruption, you can also force an entry to the debugger by entering the DEBUG command.

There are some other DCL commands you can enter that have no direct effect on the image. After using them, you can resume the execution of the image with the DCL command CONTINUE. For example:

```
$ RUN APPLIC
^Y
$ SHOW TRANSLATION INFILE
  INFILE =      (undefined)
$ DEFINE INFILE DBA1:[TESTFILES]JANUARY.DAT
$ CONTINUE
```

For a complete list of the commands you can enter following a <CTRL/Y> interruption without affecting the current image, see the *VAX/VMS Command Definition Utility Reference Manual*.

As noted above, you may use <CTRL/C> to interrupt your program; in most cases, the effect of <CTRL/C> and <CTRL/Y> is the same. However, some programs (including programs you may write) establish particular actions to take to respond to <CTRL/C>. If a program has no <CTRL/C> handling routine, then <CTRL/C> is the same as <CTRL/Y>.

4.2.4 Returning Status Values to the Command Interpreter

If you run your program as part of a command procedure, it is frequently useful to return a status value to the command procedure indicating whether the program actually executed properly. To return such a status value, call the EXIT system subroutine rather than terminating execution with a STOP, RETURN, or END statement. The EXIT subroutine can be called from any executable program unit. It terminates your program and returns the value of the argument as the return status value of the program. See Section D.4.4 for a description of the EXIT subroutine.

When the command interpreter receives a status value from a terminating program, it attempts to locate a corresponding message in a central system message file or a user-defined message file. Every possible message that can be issued by a system program, command, or component, has a unique 32-bit numeric value associated with it. These 32-bit numeric values are called condition symbols. Condition symbols are described in Section 6.9 of the *VAX FORTRAN User's Guide*.

The command interpreter does not display messages on completion of a program under the following circumstances:

- The EXIT argument specifies the value 1, corresponding to SUCCESS.
- The program does not return a value. If the program terminates with a RETURN, STOP, or END statement, a value of 1 is always returned and no message is displayed.

4.3 Finding and Correcting Run-Time Errors

Both the compiler and the VAX Run-Time Library include facilities for detecting and reporting errors. You can use the VAX Symbolic Debugger and the traceback facility to help you locate errors that occur during program execution.

4.3.1 Effects of Error-Related Command Qualifiers

At each step in compiling, linking, and executing your program, you can specify command qualifiers that affect how errors are processed.

- At compile time, you can specify the `/DEBUG` qualifier on the FORTRAN command to ensure that symbolic information is created for use by the debugger.
- At link time, you can also specify the `/DEBUG` qualifier on the LINK command to make the symbolic information available to the debugger.
- At run time, you can specify the `/DEBUG` qualifier on the RUN command to invoke the debugger.

Table 4-2 summarizes the `/DEBUG` and `/TRACEBACK` qualifiers.

Table 4-2: `/DEBUG` and `/TRACEBACK` Qualifiers

Qualifier	Command	Effect	Default
<code>/DEBUG</code>	FORTRAN	The FORTRAN compiler creates symbolic data debugger.	<code>/DEBUG=(NOSYMBOLS,TRACEBACK)</code>
<code>/DEBUG</code>	LINK	Symbolic data created by the FORTRAN compiler is passed to the debugger.	<code>/NODEBUG</code>
<code>/TRACEBACK</code>	LINK	Traceback information is passed to the debugger. Traceback will be produced.	<code>/TRACEBACK</code>
<code>/DEBUG</code>	RUN	Invokes the debugger. The <code>DBG></code> prompt will be displayed. Not needed if <code>\$ LINK/DEBUG</code> was specified.	
<code>/NODEBUG</code>	RUN	If <code>/DEBUG</code> was specified in the LINK command, <code>RUN/NODEBUG</code> starts program execution without first invoking the debugger.	

If these qualifiers are not specified at any point in the compile-link-execute sequence, a traceback list is generated by default if an execution error occurs.

To perform symbolic debugging, you must use the /DEBUG qualifier with both the FORTRAN and LINK commands, but you do not need to specify it with the RUN command. If /DEBUG is omitted from either the FORTRAN or LINK command, you can still use it with the RUN command to invoke the debugger. However, any debugging you perform must then be done by specifying virtual addresses rather than symbolic names.

If you linked your program with the debugger, but wish to execute the program without debugger intervention, specify:

```
$ RUN/NODEBUG program-name
```

If you specify LINK/NOTRACEBACK, you receive no traceback in the event of errors. A sample source program and a traceback are shown in Figure 4-1.

```
0001          PROGRAM TRACE_TEST
0002          I = 1
0003
0004          CALL SUB1(I)
0005          END
```

```
0001          SUBROUTINE SUB1(I)
0002          I = I + 1
0003          CALL SUB2
0004          RETURN
0005          END
```

```
0001
0002          SUBROUTINE SUB2
0003          COMPLEX W
0004          COMPLEX Z
0005
0006          DATA W/(0.,0.)/
0007          Z = LOG(W)
0008          RETURN
0009          END
```

```
%MTH-F-INVARGMAT, invalid argument to math library
  user PC 000034D4
```

```
%TRACE-F-TRACEBACK, symbolic stack dump follows
```

module name	routine name	line	relative PC	absolute PC
			00001368	00001368
			00002C51	00002C51
			000034D4	000034D4
SUB2	SUB2	7	00000011	00000439
SUB1	SUB1	3	0000000C	00000424
TRACE_TEST	TRACE_TEST	4	00000014	00000414

Figure 4-1: Sample FORTRAN Program and Traceback

The traceback is interpreted as follows:

When the error condition is detected, you receive the appropriate message, followed by the traceback information. The Run-Time Library displays a message indicating the nature of the error and the address at which the error occurred (user PC). This is followed by the traceback information, which is presented in inverse order to the calls. Note that values can be produced for relative and absolute PC, with no corresponding values for routine name and line. These PC values reflect procedure calls internal to the Run-Time Library.

Of particular interest are the values listed under "routine name" and "line." The names under "routine name" show what routine or subprogram called the Run-Time Library, which subsequently reported the error. The value given for "line" corresponds to the compiler-generated line number in the source program listing (not to be confused with editor-generated line numbers). With this information, you can usually isolate the error in a short time.

If you specify either LINK/DEBUG or RUN/DEBUG, the debugger assumes control of execution and you do not receive a traceback list if an error occurs. To display traceback information, you can use the debugger command SHOW CALLS.

You should specify the /NOOPTIMIZE qualifier on the FORTRAN command line whenever you use the debugger; see Section 3.2.3.15.

Chapter 5

Introduction to VAX FORTRAN

This chapter contains information on the following topics:

- The standards that VAX FORTRAN adheres to (Section 5.1)
- The elements that make up a VAX FORTRAN source program (Section 5.2)
- The character set supported by VAX FORTRAN (Section 5.3)
- The general rules for coding in VAX FORTRAN (Section 5.4)

5.1 VAX FORTRAN Language Definition

VAX FORTRAN is based on American National Standard FORTRAN-77 (ANSI X3.9-1978). It includes support for programs that conform to the previous standard (ANSI X3.9-1966). Extensions to the FORTRAN-77 standard are printed in blue in this manual.

VAX FORTRAN provides the following extensions to the ANSI standard:

- Relative file organization
- Indexed file organization
- Conformance with the VAX procedure-calling standard
- Records and structures
- DO WHILE statement
- Additional data types
- Namelist-directed input/output
- Hexadecimal constants and field descriptors
- Symbolic debugging facility

VAX FORTRAN is also a compatible superset of PDP-11 FORTRAN-77. This means that you can compile existing PDP-11 FORTRAN-77 source programs using the VAX FORTRAN compiler (see Appendix B in the *VAX FORTRAN User's Guide*).

5.2 Elements of FORTRAN Source Programs

This section provides an overview of the make up of a FORTRAN source program. It describes the concept of a program unit and the rules governing the use of statements and symbols within a program unit. It also describes the use of comments within programs.

5.2.1 Program Units

A program unit is a sequence of statements that defines a computing procedure and is terminated by an END statement. A program unit can be either a main program or a subprogram. An executable program consists of one main program and, optionally, one or more subprograms.

A subprogram is a program unit that is separate from the main program. Subprograms are invoked from the main program or another subprogram. There are two types of subprograms: function subprograms and subroutine subprograms. See Chapter 10 for detailed information on subprograms.

5.2.2 Statements

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

Statements are divided into physical sections called lines. A line is a string of up to 80 characters (optionally, 132; see Section 5.4.5). (Note: FORTRAN-77 limits the length to 72 characters.) If a statement is too long to fit on one line, you can continue it on one or more additional lines called continuation lines. A continuation line is identified by a continuation character in the sixth column of that line. (For further information on continuation characters, see Section 5.4.4.)

You can identify a statement with a statement label so that other statements can refer to it, either to get the information it contains or to transfer control to it. A statement label must be an integer, and it must appear in the first five columns of a statement's initial line. Any statement can have a label; however, you can only refer to labels on executable statements and FORMAT statements.

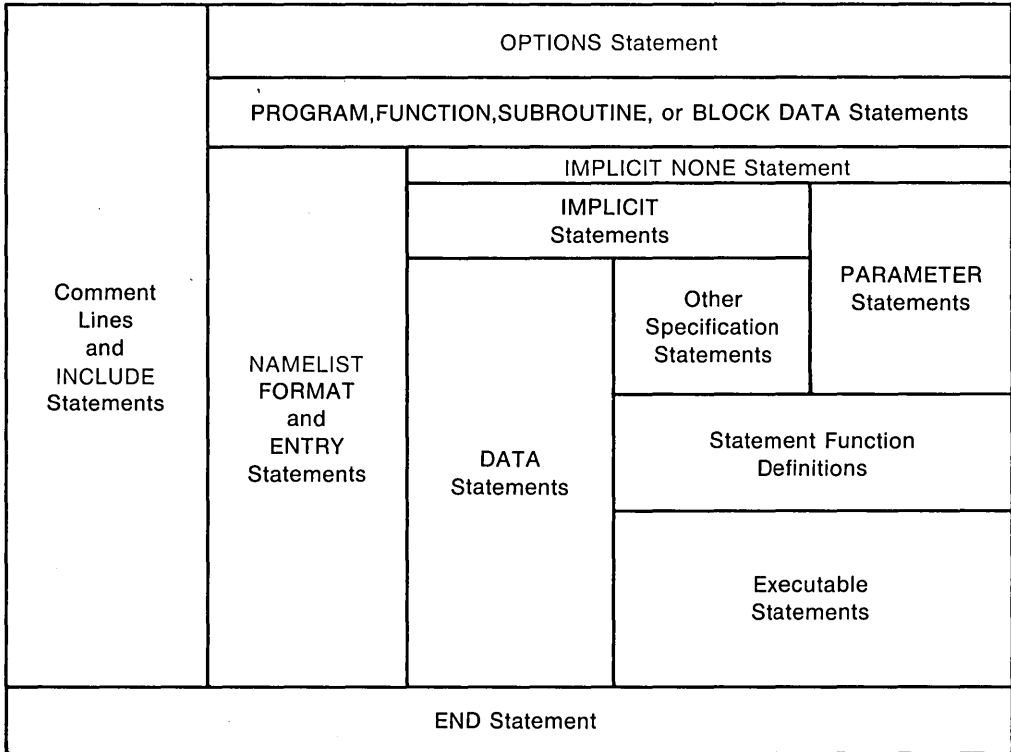
5.2.2.1 Order of Statements in a Program Unit

Figure 5-1 shows the required order of statements in a FORTRAN program unit. In this figure, vertical lines separate statement types that can be interspersed. For example, DATA statements can be interspersed with executable statements. On the other hand, horizontal lines indicate statement types that cannot be interspersed. For example, type declaration statements cannot be interspersed with executable statements.

Statements included in the category of “executable statements” in Figure 5-1 are: ACCEPT, ASSIGN, assignment statements, BACKSPACE, CALL, CLOSE, CONTINUE, DELETE, DO and END DO, ELSE, END, ENDFILE, FIND, GO TO (normal, computed, and assigned), IF (arithmetic, logical, and block) and END IF, INQUIRE, OPEN, PAUSE, PRINT, READ, RETURN, REWIND, REWRITE, STOP, TYPE, UNLOCK, and WRITE.

Statements included in the category of “other specification statements” in Figure 5-1 are: COMMON, DICTIONARY, DIMENSION, EQUIVALENCE, EXTERNAL, INTRINSIC, RECORD, SAVE, structure declarations, type declarations, and VOLATILE. (Note: The statements STRUCTURE and END STRUCTURE, UNION and END UNION, and MAP and END MAP are included in this category. They are used only in structure declaration blocks.)

As a VAX FORTRAN extension, DATA statements can be freely interspersed with PARAMETER statements and other specification statements.



ZK-615-82

Figure 5-1: Required Order of Statements and Lines

5.2.3 Symbolic Names

Symbolic names identify entities within a FORTRAN program unit. These entities are listed in Table 5-2.

A symbolic name is a string of letters, digits, and the special characters dollar sign (\$) and underscore (_). The first character in a symbolic name must be a letter. The symbolic name can contain a maximum of 31 characters. (Note: FORTRAN-77 limits the length of a symbolic name to six characters.)

Examples of valid symbolic names are:

```
NUMBER  
K9  
X  
FIND_IT
```

Examples of invalid symbolic names are:

```
5Q           (begins with a numeral)  
B.4         (contains a special character other than _ or $)  
$FREQ      (begins with $)
```

By convention, symbolic names containing a dollar sign (\$) are reserved for use in DIGITAL-supplied software components. To avoid name conflicts, you should not define any symbolic names in your program that contain a dollar sign.

In most cases, you cannot use the same symbolic name to identify two or more entities in the same program unit. The exceptions are as follows:

- The names of structures can be used as the names of fields of records (see Sections 8.15.1 and 8.15.2, respectively).
- Common block names can be used as variable or array names.

In an executable program consisting of two or more program units, the symbolic names of the following entities must be unique within the entire program:

- Function subprograms
- Subroutine subprograms
- Common blocks
- Main programs
- Block data subprograms
- Function entry points
- Subroutine entry points

That is, if your program contains a function named BTU, you cannot use BTU as the symbolic name of any other subprogram, entry, or common block in the program, even if the name appears in a different program unit.

Table 5-1 lists those entities that can be given a symbolic name. It also indicates whether the entities can be given a data type. Sections 6.2.2.1 and 6.2.2.2 discuss how to specify the data type of a symbolic name.

Table 5-1: Entities Identified by Symbolic Names

Entity	Typed
Variables	Yes
Arrays	Yes
Structures	No
Records	No
Record elements	Yes
Statement functions	Yes
Intrinsic functions	Yes
Function subprograms	Yes
Subroutine subprograms	No
Common blocks	No
Namelist data groups	No
Main programs	No
Block data subprograms	No
Function entry points	Yes
Subroutine entry points	No
Parameter constants	Yes

5.2.4 Comments

Comments do not affect program processing in any way. They are merely a documentation aid to the programmer. You can use them freely to describe the actions of the program, to identify program sections and processes, and to provide greater ease in reading the source program listing. The letter C or an asterisk (*) in the first column of a source line identifies that line as a comment; a line containing only spaces is also a comment line. In addition, if you place an exclamation point (!) in column 1 or anywhere in the statement portion of a source line, the remainder of that line is treated as a comment.

5.3 FORTRAN Character Set

The character set supported by VAX FORTRAN consists of the following:

- All uppercase and lowercase letters (A through Z, a through z)
- The numerals 0 through 9
- The following special characters:

Character	Name
△ or <TAB>	Space or tab
=	Equal sign
+	Plus sign
-	Minus sign
*	Asterisk
/	Slash
(Left parenthesis
)	Right parenthesis
,	Comma
.	Period
'	Apostrophe
"	Quotation mark
\$	Dollar sign
—	Underscore
!	Exclamation point
:	Colon
<	Left angle bracket
>	Right angle bracket
%	Percent sign
&	Ampersand

You can use the space character to improve the legibility of a FORTRAN statement. The compiler ignores all spaces in a statement field except those within a character or Hollerith constant. For example, GO TO and GOTO are equivalent.

Other printable ASCII characters can appear in a FORTRAN statement only as part of a character or Hollerith constant (see Appendix B for a list of printable characters). Any printable character can appear in a comment. Nonprintable characters should not be used in a FORTRAN source statement; if they are used, they appear as question marks.

Except in character and Hollerith constants, the compiler makes no distinction between uppercase and lowercase letters.

5.4 Format Requirements of FORTRAN Source Code

Each FORTRAN line has the following four fields:

- Statement label field
- Continuation indicator field
- Statement field
- Sequence number field

There are two ways to code a FORTRAN line: fixed format or tab format. You may prefer to use the fixed format method when punching cards or using a coding form. The tab format method is convenient when you are entering lines at a terminal with a text editor.

5.4.1 Fixed-Format Lines

As shown in Figure 5-2, a FORTRAN line is divided into fields for statement labels, continuation indicators, statement text, and sequence numbers. Each column represents a single character. Sections 5.4.3 through 5.4.6 describe the use of each field.

FORTRAN
CODING FORM

CODER	DATE	PAGE
PROBLEM		

C Comment		FORTRAN STATEMENT																																																																												IDENTIFICATION		
STATEMENT NUMBER	Continuation																																																																															
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	
	C	THIS PROGRAM CALCULATES PRIME NUMBERS FROM 11 TO 50																																																																														
		DO 10, I=11, 50, 2																																																																														
		J=1																																																																														
4		J=J+2																																																																														
		A=J																																																																														
		A=1/A																																																																														
		L=1/J																																																																														
		B=A-L																																																																														
		IF (B) 5, 10, 5																																																																														
5		IF (J.LT..5*QRT (FLOAT (I))) GO TO 4																																																																														
		TYPE 105, I																																																																														
10		CONTINUE																																																																														
105		FORMAT (I4, ' IS PRIME:')																																																																														
		END																																																																														

PG-3

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

ZK-613-82

Figure 5-2: FORTRAN Coding Form

To enter an item in a field, enter it in the column(s) in the coding form, as listed below:

Field	Column(s)
Statement label	1 through 5
Continuation indicator	6
Statement	7 through 72 (optionally, to 132)
Sequence number	73 through 80

5.4.2 Tab-Format Lines

You can specify the statement label field, the continuation indicator field, and the statement field using tab formatting. However, you cannot specify a sequence number field using this method of coding. Figure 5-3 illustrates FORTRAN lines coded using tab formatting and the equivalent lines with fixed formatting.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C							F	I	R	S	T		V	A	L	U	E			
10	0						I	=	J		+	5	*	K		+				
					1		L	*	M											
							I	V	A	L	=	I	+	2						

ZK-614-82

Figure 5-3: Line Formatting Example

The statement label field consists of the characters that you type before the first tab character. The statement label field cannot have more than five characters.

After you type the first tab character, you can type either the continuation indicator field or the statement field.

To enter the continuation indicator field, type any nonzero digit after the first tab. If you enter the continuation indicator field, the statement field consists of all the characters after the digit to the end of the line.

To enter the statement field without a continuation indicator field, type the statement immediately after the first tab. Note that no FORTRAN statement starts with a digit.

Many text editors and terminals advance the terminal print carriage to a predefined print position when you press the TAB key. However, this action is not related to the FORTRAN compiler's interpretation of the tab character described above.

The compiler treats the tab character in a statement field the same as a space. In the source listing that the compiler produces, the tab causes the character that follows to be printed at the next tab stop (located at columns 9, 17, 25, 33, and so on).

NOTE

Do not use tabs when you are using sequence numbers. If you use tabs to position your sequence numbers, the compiler may interpret the sequence numbers as part of the statement fields in your program.

5.4.3 Statement Label Field

Any statement can have a label. A statement label or statement number consists of from one to five decimal digits in the statement label field of a statement's initial line. Spaces and leading zeros are ignored. An all-zero statement label is invalid.

The only statements that can be referred to by other statements are labeled **FORMAT** statements and labeled executable statements (see Section 5.2.2). **FORMAT** statements are referred to only in the format specifier of an I/O statement or in an **ASSIGN** statement. No two statements within a program unit can have the same label.

The first column of the label field can contain two special indicators: the comment indicator and the debugging statement indicator. These indicators are described in Sections 5.4.3.1 and 5.4.3.2.

The statement label field of a continuation line must be blank—except in the case of a debugging statement (see Section 5.4.3.2).

5.4.3.1 Comment Indicator

You can use the letter **C**, an asterisk (*), or an exclamation point (!) in column 1 to indicate that a line is a comment. The compiler prints that line in the source program listing and then ignores the line. An all-blank line is also considered to be a comment. The exclamation point can also be used anywhere in the statement field (except when used in a Hollerith or character constant) to start an end-of-line comment.

5.4.3.2 Debugging Statement Indicator

You can use the letter **D** in column 1 to designate debugging statements. The initial line of the debugging statement can contain a statement label in the remaining columns of the label field. If a debugging statement is continued onto more than one line, every continuation line must begin with a **D** (in column 1) and a continuation indicator.

The compiler treats the debugging statement either as source text to be compiled or as a comment, depending on the setting of the **/D__LINES** compiler command qualifier. If you specify **/D__LINES**, debugging statements are compiled as a part of the source program. If you do not specify **/D__LINES**, debugging statements are treated as comments.

5.4.4 Continuation Indicator Field

A continuation indicator is any character, except zero or space, in column 6 of a FORTRAN line, or any digit, except zero, after the first tab. The compiler considers the characters after the continuation character to be the characters following the last character of the previous line, as if there were no break at that point. If a continuation indicator is a zero or space, the compiler considers the line to be an initial line of a FORTRAN statement.

Comment lines cannot be continued. They can occur between a statement's initial line and its continuation line(s), or between successive continuation lines.

5.4.5 Statement Field

The text of a FORTRAN statement is placed in the statement field. Because the compiler ignores the tab character and spaces (except in character and Hollerith constants), you can space the text in any way desired for maximum legibility.

By default, the statement field extends to character position 72. If the default is in effect, any text following position 72 is ignored and no warning message is printed. However, if the `/EXTEND__SOURCE` qualifier is specified on the FORTRAN command (see Section 3.2.3.7), the statement field is extended to position 132. Any text beyond that position will generate a fatal error and will cause immediate termination of the compilation.

5.4.6 Sequence Number Field

By default, a sequence number or other identifying information can appear in columns 73 through 80 of any line in a FORTRAN program. The compiler ignores the characters in this field. However, if the `/EXTEND__SOURCE` qualifier is specified on the FORTRAN command (see Section 3.2.3.7), a sequence number field does not exist because the statement field is extended to position 132.

Chapter 6

Data Types, Data Items, and Expressions

This chapter contains information on the following topics:

- Data types—integer, real, complex, logical, character, and Hollerith (Section 6.1)
- Data items—constants, variables, arrays, character substrings, and records (Section 6.2)
- Expressions—arithmetic, character, relational, and logical (Section 6.3)

6.1 Data Types

Each constant, variable, array, expression, or function reference in a FORTRAN statement represents typed data. The data type of these items can be inherent in their constructions, implied by convention, or explicitly declared. The data types available in FORTRAN, and their definitions, are:

- Integer—a whole number.
- REAL (REAL*4)—a floating point number, that is, a whole number, a decimal fraction, or a combination of the two.
- DOUBLE PRECISION (REAL*8)—similar to REAL*4; has more than twice the degree of accuracy in its representation (the G_floating implementation also has an extended range).
- REAL*16—similar to REAL*4; has an extended range, and more than four times the accuracy in its representation.
- COMPLEX (COMPLEX*8)—a pair of REAL*4 values that represent a complex number; the first value represents the real part of that number, and the second represents the imaginary part.
- DOUBLE COMPLEX (COMPLEX*16)—similar to complex; its real and imaginary parts are REAL*8.

- Logical—a logical value, `.TRUE.` or `.FALSE.`
- Character—a string of characters.
- Hollerith—a string of printable characters preceded by a character count and the letter H.

See Appendix C for descriptions of the VAX hardware representations of these data types.

6.1.1 Storage Requirements

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

ANSI FORTRAN defines a “numeric storage unit” as the amount of storage needed to represent a `REAL`, `INTEGER`, or `LOGICAL` value. In VAX FORTRAN, a numeric storage unit corresponds to four bytes of memory. `REAL*8` and `COMPLEX*8` values occupy two of these numeric storage units, whereas `REAL*16` and `COMPLEX*16` values occupy four.

ANSI FORTRAN defines a “character storage unit” as the amount of storage needed to represent one character value. In VAX FORTRAN, a character storage unit corresponds to one byte of memory.

VAX FORTRAN provides additional data types for optimum selection of performance and memory requirements. Table 6-1 lists the data types available, the names associated with each data type, and the amount of storage required (in bytes). The form `*n` appended to a data type name is called a data type length specifier.

Table 6-1: Data Type Storage Requirements

Data Type	Storage Requirements (in bytes)
BYTE	1 ¹
LOGICAL	2 or 4 ²
LOGICAL*1	1 ¹
LOGICAL*2	2
LOGICAL*4	4
INTEGER	2 or 4 ²
INTEGER*2	2
INTEGER*4	4
REAL	4
REAL*4	4
REAL*8	8
DOUBLE PRECISION	8
REAL*16	16

Table 6-1 (Cont.): Data Type Storage Requirements

Data Type	Storage Requirements (in bytes)
COMPLEX	8
COMPLEX*8	8
COMPLEX*16	16
DOUBLE COMPLEX	16
CHARACTER*len	len ³
CHARACTER*(*)	

¹ The 1-byte storage area can contain the logical values true or false, a single character, or integers in the range -128 to 127.

² Either two or four bytes are allocated depending on the setting of the [NOI]4 qualifier. The default allocation is four bytes.

³ The value of len is the number of characters specified, which can be in the range 1 to 65535. Passed-length format, *(*), applies to dummy arguments or character functions, and indicates that the length of the actual argument or function is used (see Chapter 10).

6.1.2 VAX Implementations of REAL*8

There are two implementations of the REAL*8 (and COMPLEX*16) data type on VAX: D__floating and G__floating. The G__floating implementation offers a greater range, but a smaller number of significant digits of precision, than the D__floating implementation. You select the G__floating implementation by compiling the program with the /G__FLOATING qualifier in the FORTRAN command line or the OPTIONS statement; the default implementation of REAL*8 is D__floating.

Some VAX processors do not implement one or more of the floating point data types. For these processors, the data types not supported in hardware or microcode are emulated in software. You should be aware of which data types are emulated on the system you use because processing time with software emulation is much slower. The FORTRAN data type you use—especially REAL*8—should be chosen with this in mind.

See Sections 6.2.1.2, C.4, C.4.2, and C.4.3 for more detailed information on the two implementations of the REAL*8 data type.

6.2 Data Items

Data items that you use in VAX FORTRAN statements are as follows:

- Constants—fixed, self-describing values.
- Variables—stored values represented by symbolic names.
- Arrays—groups of values that are stored contiguously and can be referred to individually or collectively. Individual values are called array elements.
- Character substrings—a contiguous segment of a character variable or character array element.
- Records—structured data items consisting of one or more elements (variables and arrays) or one or more groups of these elements. Different record elements in the same record can have unlike data types.

These data items are discussed in this order in Sections 6.2.1 through 6.2.5.

6.2.1 Constants

A constant is a data item with a fixed value; the value cannot be changed during program execution. The value of a constant can be a numeric value, a logical value, or a character string. There are eight types of constants:

- Integer
- Real
- Complex
- Octal
- Hexadecimal
- Logical
- Character
- Hollerith

Octal, hexadecimal, and Hollerith constants have no data type. They assume a data type that conforms to the context in which they appear (see Sections 6.2.1.4 and 6.2.1.7, respectively).

NOTE

The generic term *scalar reference* is used throughout this manual to refer collectively to all references that resolve to single, typed data items. All eight types of constants fall into the scalar reference category.

Section 6.2.6 provides a thorough discussion of this new terminology.

6.2.1.1 Integer Constants

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

An integer constant has the form:

`snn`

where:

`s`

is an optional sign.

`nn`

is a string of decimal digits.

Leading zeros, if any, are ignored. A minus sign must appear before a negative integer constant, whereas a plus sign is optional before a positive constant (an unsigned constant is assumed to be positive). Except for a leading algebraic sign, an integer constant cannot contain any character other than the numerals 0 through 9. The value of an integer constant must be within the range -2147483648 to 2147483647.

Examples of valid and invalid integer constants are:

Valid	Invalid (with explanation)
<code>0</code>	<code>999999999999</code> (too large)
<code>-127</code>	<code>3.14</code> (decimal point and
<code>+32123</code>	<code>32,767</code> comma not allowed)

If the value of the constant is within the range -32768 to 32767, it represents a 2-byte signed quantity and is treated as an `INTEGER*2` data type. If the value is outside that range, it represents a 4-byte signed quantity and is treated as an `INTEGER*4` data type.

Integer constants can also be specified in octal form; see Section A.5.

6.2.1.2 Real Constants

A real constant is a number written with a decimal point or exponent (or both). The constant can be positive, zero, or negative, and can have single precision (`REAL*4`), double precision (`REAL*8`), or quad precision (`REAL*16`).

The different types of real constants (`REAL*4`, `REAL*8`, and `REAL*16`) are described under the headings that follow.

REAL*4 (REAL) Constants

A REAL*4 constant can be any one of the following:

- A basic real constant
- A basic real constant followed by a decimal exponent
- An integer constant followed by a decimal exponent

Integer constants are defined in the preceding subsection. A basic real constant is a string of decimal digits having one of the following forms:

s.nn
snn.nn
snn.

where:

s
is an optional sign.

nn
is a string of decimal digits.

The decimal point can appear anywhere in the string. The number of digits is not limited, but typically only the leftmost seven digits are significant. Leading zeros (zeros to the left of the first nonzero digit) are ignored in counting the leftmost seven digits. Thus, in the constant 0.00001234567, all of the nonzero digits, and none of the zeros, are significant.

A decimal exponent has the form:

Esnn

where:

s
is an optional sign.

nn
is an integer constant.

The exponent represents a power of 10 by which the preceding real or integer constant is to be multiplied (for example, 1.0E6 represents the value $1.0 * 10^{**6}$).

A REAL*4 constant occupies four bytes of VAX storage, and it is interpreted as a real number with a degree of precision that is typically seven decimal digits (see Sections C.4 and C.4.1).

A minus sign must appear before a negative REAL*4 constant; a plus sign is optional before a positive constant. Similarly, a minus sign must appear between the letter E and a negative exponent, whereas a plus sign is optional between the letter E and a positive exponent.

A REAL*4 constant cannot contain any character other than the numerals 0 through 9, except for algebraic signs, a decimal point, and the letter E (if used).

If the letter E appears in a REAL*4 constant, an integer constant exponent field must follow. The exponent field cannot be omitted; it can, however, be zero.

The magnitude of a nonzero REAL*4 constant cannot be smaller than approximately 0.29E-38 or greater than approximately 1.7E38.

Examples of valid and invalid REAL*4 constants follow:

Valid	Invalid (with explanation)
3.14159	1,234,567. (commas not allowed)
621712.	325E-45 (too small)
-.00127	-47.E47 (too large)
+5.0E3	100 (decimal point missing)
2E-3	\$25.00 (special character not allowed)

REAL*8 (DOUBLE PRECISION) Constants

A REAL*8 constant is a basic real constant or an integer constant followed by a decimal exponent of the form:

Dsnn

where:

s

is an optional sign.

nn

is a string of decimal digits.

There are two implementations of the REAL*8 constant: D__floating and G__floating. Both implementations have the same syntax and storage requirements, but each has a different number of significant digits and a different exponent range. The G__floating implementation is invoked with the /G__FLOATING qualifier.

A REAL*8 constant occupies eight bytes of storage. The number of digits that precede the exponent in a REAL*8 constant is not limited. However, the degree of precision is typically only the leftmost 16 (for D__floating) or 15 (for G__floating) digits (see Sections C.4, C.4.2, and C.4.3).

A minus sign must appear before a negative REAL*8 constant; a plus sign is optional before a positive constant. Similarly, a minus sign must appear between the letter D and a negative exponent, whereas a plus sign is optional between the letter D and a positive exponent.

If the letter D appears in a REAL*8 constant, an integer constant exponent field must follow. The exponent field following the letter D cannot be omitted; it can, however, be zero.

The magnitude of a nonzero REAL*8 constant cannot be less than approximately 0.29D-38 or greater than approximately 1.7D38 for the D__floating implementation; nor can it be less than approximately 0.56D-308 or greater than approximately 0.9D308 for the G__floating implementation.

Examples of valid and invalid D__floating and G__floating REAL*8 constants follow:

D__floating		
Valid	Invalid (with explanation)	
1234567890D+5	1234567890D45	(too large)
+2.71828182846182D00	1234567890.0D-89	(too small)
-72.5D-15 1D0	+2.7182812846182	(no Dsnn present; this is a valid single-precision constant)

G__floating		
Valid	Invalid(with explanation)	
123456789.D0	123456789.D400	(too large)
+2.34567890123D-5 -1D+300	123456789.D-400	(too small)

6.2.1.2.1 REAL*16 Constants

A REAL*16 constant is a basic real constant or an integer constant followed by a decimal exponent of the form:

Qsnn

where:

s

is an optional sign.

nn

is a string of decimal digits.

A REAL*16 constant occupies 16 bytes of VAX storage. The number of digits that precede the exponent is not limited; however, typically only the leftmost 33 digits are significant (see Sections C.4 and C.4.4).

A minus sign must appear before a negative REAL*16 constant; a plus sign is optional before a positive constant. Similarly, a minus sign is required between the letter Q and a negative exponent, whereas a plus sign is optional between the letter Q and a positive exponent.

If the letter Q appears in a REAL*16 constant, an integer constant exponent field must follow. The exponent field following the letter Q cannot be omitted; however, it can be zero.

The magnitude of a nonzero REAL*16 constant cannot be less than approximately 0.84Q-4932 or greater than approximately 0.59Q4932.

Examples of valid and invalid REAL*16 constants follow:

Valid	Invalid (with explanation)
123456789Q4000	1.Q5000 (too large)
-1.23Q-400	1.Q-5000 (too small)
+2.72Q0	

6.2.1.3 Complex Constants

A complex constant consists of a pair of real or integer constants. The two constants are separated by a comma and enclosed in parentheses. The first constant represents the real part of that number, and the second constant represents the imaginary part.

VAX FORTRAN supports COMPLEX*8 and COMPLEX*16 complex constants. These are described under the headings that follow.

COMPLEX*8 (COMPLEX) Constants

A COMPLEX*8 constant is a pair of integer or REAL*4 constants that represents a complex number.

A COMPLEX*8 constant has the form:

(c,c)

where:

c

is an integer or REAL*4 constant.

The parentheses and comma are part of the constant and are required. See Section 6.2.1.2 for the rules for forming REAL*4 constants.

A COMPLEX*8 constant occupies eight bytes of VAX storage and is interpreted as a complex number (see Sections C.4 and C.4.5).

The following are examples of valid and invalid COMPLEX*8 constants.

Valid	Invalid (with explanation)
(1.7039,-1.70391)	(1.23,) (second REAL constant is missing)
(+12739E3,0.)	
(1,2)	(1.0, 000) (REAL*16 constants are not allowed)

COMPLEX*16 (DOUBLE COMPLEX) Constants

A COMPLEX*16 constant is a pair of constants that represent a complex number. One constant must be REAL*8; the other must be an integer, REAL*4, or REAL*8. The two constants are separated by a comma and enclosed in parentheses; the first constant represents the real part of the complex number, the second the imaginary part. There are two implementations of COMPLEX*16, corresponding to the D_floating and G_floating implementations of REAL*8.

A COMPLEX*16 constant has the form:

(c,c)

where:

c
is an integer, a REAL*4, or a REAL*8 constant. (One of the pair must be a REAL*8 constant.)

The parentheses and the comma are part of the constant and are required. See Section 6.2.1.2 for the rules governing the formation of REAL*8 constants.

A COMPLEX*16 constant occupies 16 bytes of VAX storage and is interpreted as a complex number (see Sections C.4, C.4.6, and C.4.7).

Examples of valid and invalid COMPLEX*16 constants follow:

Valid	Invalid (with explanation)
(1.7039D0,-1.7039D0)	(1.23D0) (second constant missing)
(+12739D3,0.D0)	(0.8Q0,0.4Q0) (REAL*16 constants not allowed)
	(1.0D300,-1.0D300) (both constants out of range for D_floating implementation of REAL*8; valid for G_floating implementation of REAL*8)

6.2.1.4 Octal and Hexadecimal Constants

Octal and hexadecimal constants are alternative ways to represent numeric constants. You can use them wherever numeric constants are allowed.

An octal constant is a string of octal digits enclosed by apostrophes and followed by the alphabetic character O. An octal constant has the form:

$$'c_1c_2c_3\dots c_n'O$$

where:

c

is a digit in the range 0 to 7.

A hexadecimal constant is a string of digits enclosed by apostrophes and followed by the alphabetic character X. A hexadecimal constant has the form:

$$'c_1c_2c_3\dots c_n'X$$

where:

c

is a digit in the range 0 to 9, or a letter in the range A to F or a to f.

Leading zeros are ignored in octal and hexadecimal constants. You can specify up to 128 bits (43 octal digits, 32 hexadecimal digits).

Examples of valid and invalid octal constants are:

Valid	Invalid (with explanation)
'07737'O	'7782'O (invalid character)
'1'O	7772'O (no initial apostrophe)
	'0737' (no O after second apostrophe)

Examples of valid and invalid hexadecimal constants are:

Valid	Invalid (with explanation)
'AF9730'X	'999.'X (invalid character)
'FFABC'X	'F9X (no apostrophe before the X)

Octal and hexadecimal constants are “typeless” numeric constants. They assume data types based on the way they are used (and thus are not converted before use), as follows:

- When the constant is used with a binary operator, including the assignment operator, the data type of the constant is the data type of the other operand. For example:

Statement	Data Type of Constant	Length of Constant
INTEGER*2 ICOUNT		
REAL*8 DOUBLE		
RAPHA = '99AF2'X	REAL*4	4
JCOUNT = ICOUNT + '777'0	INTEGER*2	2
DOUBLE = 'FFF99A'X	REAL*8	8
IF (N .EQ. '123'0 GO TO 10)'	INTEGER*4	4

- When a specific data type (generally integer) is required, that type is assumed for the constant. For example:

Statement	Data Type of Constant	Length of Constant
Y(IX) = Y('15'0) + 3,	INTEGER*4	4

- When the constant is used as an actual argument, no data type is assumed; however, a length of four bytes is always used. For example:

Statement	Data Type of Constant	Length of Constant
CALL APAC('34BC2'X)	None	4

- When the constant is used in any other context, an INTEGER*4 data type is assumed. For example:

Statement	Data Type of Constant	Length of Constant
IF ('AF77'X) 1,2,3	INTEGER*4	4
I = '7777'0 - 'A39'X	INTEGER*4	4
J = .NOT. '73777'0	INTEGER*4	4

An octal or hexadecimal constant specifies as much as 16 bytes of data. When the data type implies that the length of the constant is more than the number of digits specified, the leftmost digits have a value of zero. When the data type implies that the length of the constant is less than the number of digits specified, the constant is truncated on the left. An error results if any nonzero digits are truncated. Table 6-1 (in Section 6.1.1) lists the number of bytes that each data type requires.

6.2.1.5 Logical Constants

A logical constant specifies a logical value, true or false. Thus, only the following two logical constants are possible:

.TRUE.

.FALSE.

The delimiting periods are a required part of each constant.

6.2.1.6 Character Constants

A character constant is a string of printable ASCII characters enclosed by apostrophes.

A character constant has the form:

'c₁c₂c₃...c_n'

where:

c

is a printable character.

Both delimiting apostrophes must be present.

The value of a character constant is the string of characters between the delimiting apostrophes. The value does not include the delimiting apostrophes, but does include all spaces or tabs within the apostrophes.

Within a character constant, the apostrophe character is represented by two consecutive apostrophes (with no space or other character between them).

The length of the character constant is the number of characters between the apostrophes, except that two consecutive apostrophes represent a single apostrophe. The length of a character constant must be in the range 1 to 2000.

Examples of valid and invalid character constants are:

Valid	Invalid (with explanation)
'WHAT?'	'HEADINGS (no trailing apostrophe)
'TODAY'S DATE IS: '	" (character constant must contain at least one character)
'HE SAID, "HELLO"'	"NOW/OR NEVER" (quotation marks cannot be used in place of apostrophes)

If a character constant appears in a numeric context (for example, as the expression on the right side of an arithmetic assignment statement), it is considered a Hollerith constant (see next section).

6.2.1.7 Hollerith Constants

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

A Hollerith constant has the form:

$$nHc_1c_2c_3\dots c_n$$

where:

n
is an unsigned, nonzero integer constant stating the number of characters in the string (including spaces and tabs).

c
is a printable character.

A Hollerith constant can be a string of 1 to 2000 characters.

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

Hollerith constants have no data type. They assume a numeric data type according to the context in which they are used. Hollerith constants cannot assume a character data type and cannot be used where a character value is expected.

Examples of valid and invalid Hollerith constants are:

Valid	Invalid (with explanation)
16HTODAY'S DATE IS:	3HABCD (wrong number of characters)
1HB	0H (Hollerith constants must contain at least one character)

When Hollerith constants are used in numeric expressions, they assume data types according to the following rules:

- When the constant is used with a binary operator, including the assignment operator, the data type of the constant is the data type of the other operand. For example:

Statement	Data Type of Constant	Length of Constant
INTEGER*2 ICOUNT		
REAL*8 DOUBLE		
RALPHA = 4HABCD	REAL*4	4
JCOUNT = ICOUNT + 2HXY	INTEGER*2	2
DOUBLE = 8HABCDEFGH	REAL*8	8
IF (N. EQ. 1HZ) GO TO 10	INTEGER*4	4

- When a specific data type is required, generally integer, that type is assumed for the constant. For example:

Statement	Data Type of Constant	Length of Constant
Y(IX) = Y(1HA) + 3.	INTEGER*4	4

- When the constant is used as an actual argument, no data type is assumed. For example:

Statement	Data Type of Constant	Length of Constant
CALL APAC(9HABCDEFGHI)	None	9

- When the constant is used in any other context, an INTEGER*4 data type is assumed. For example:

Statement	Data Type of Constant	Length of Constant
IF(2HAB) 1,2,3	INTEGER*4	4
I = 1HC - 1HA	INTEGER*4	4
J = .NOT. 1HB	INTEGER*4	4

When the length of the constant is less than the length implied by the data type, spaces are appended to the constant on the right. When the length of the constant is greater than the length implied by the data type, the constant is truncated on the right. An error results if any characters other than space characters are truncated.

Table 6-1 (in Section 6.1.1) lists the number of characters required for each data type. Each character occupies one byte of storage.

6.2.2 Variables

A variable is represented by a symbolic name associated with a storage location. The value of the variable is the value currently stored in that location; you can change that value by assigning a new value to the variable. (See Section 5.2.3 for the form of a symbolic name.)

Variables are classified by data type, just as constants are. The data type of a variable indicates the type of data it represents, its precision, and its storage requirements. When data of any type is assigned to a variable, it is converted, if necessary, to the data type of the variable. You can establish the data type of a variable by type declaration statements, IMPLICIT statements, or predefined typing rules.

NOTE

The generic term *scalar reference* is used throughout this manual to refer collectively to all references that resolve to single, typed data items. All types of variables fall into the scalar reference category.

Section 6.2.6 provides a thorough discussion of this new terminology.

Two or more variables are associated with each other when each is associated with the same storage location. They are partially associated when part (but not all) of the storage associated with one variable is the same as part or all of the storage associated with another variable. Association and partial association occur when you use COMMON statements, EQUIVALENCE statements, MAP declarations (within STRUCTURE declaration blocks), or actual arguments and dummy arguments in subprogram references.

A variable is considered defined if the storage associated with it contains data of the same type as that of the name. A variable can be defined before program execution by a DATA statement or during execution by an assignment or input statement.

If variables of different data types are associated (or partially associated) with the same storage location, and the value of one variable is defined (for example, by assignment), the value of the other variable becomes undefined.

6.2.2.1 Data Type by Specification

Type declaration statements (see Section 8.4) specify that given variables are to represent specified data types. For example:

```
COMPLEX VAR1  
DOUBLE PRECISION VAR2
```

These statements indicate that the variable VAR1 is to be associated with an 8-byte storage location that is to contain complex data, and that the variable VAR2 is to be associated with an 8-byte double-precision storage location.

The IMPLICIT statement (see Section 8.8) has a broader scope. It states that, in the absence of an explicit type declaration, any variable with a name that begins with a specified letter, or any letter within a specified range, is to represent a specified data type.

You can explicitly specify the data type of a variable only once. An explicit data type specification takes precedence over the type implied by an IMPLICIT statement.

Character type declaration statements (see Sections 8.4 and 8.4.2) specify that given variables are to represent character values with the length specified. For example:

```
CHARACTER*72 INLINE  
CHARACTER NAME*12, NUMBER*9
```

These statements indicate that the variables INLINE, NAME, and NUMBER are to be associated with storage locations containing character data of lengths 72, 12, and 9, respectively.

Passed-length character arguments are used within a single subprogram to process character strings of different lengths. The passed-length character argument has a length specification of asterisk (*). For example:

```
CHARACTER*(*) CHARDUMMY
```

The passed-length character argument assumes the length of the actual argument (see Chapter 10).

6.2.2.2 Data Type by Implication

In the absence of either IMPLICIT statements or explicit type statements, all variables with names beginning with I, J, K, L, M, or N are assumed to be integer variables. Variables with names beginning with any other letter are assumed to be REAL*4 variables. For example:

Real Variables	Integer Variables
ALPHA	JCOUNT
BETA	ITEM
TOTAL	NTOTAL

6.2.3 Arrays

An array is a group of contiguous storage locations associated with a single symbolic name, the array name. The individual storage locations (called array elements) are referred to by a subscript appended to the array name. Section 6.2.3.2 discusses subscripts.

An array can have from one to seven dimensions. For example, a column of figures is a one-dimensional array. A table of more than one column of figures is a two-dimensional array. To refer to a specific value in this array, you must specify both its row number and its column number. A table of figures that covers several pages is a three-dimensional array. To locate a value in this array, you must specify the row number, column number, and a page number.

The following FORTRAN statements establish arrays:

- Type declaration statements (see Section 8.4)
- The DIMENSION statement (see Section 8.5)
- The COMMON statement (see Section 8.2)

These statements contain array declarators (see Section 6.2.3.1) that define the name of the array, the number of dimensions in the array, and the number of elements in each dimension.

An element of an array is considered defined if the storage associated with it contains data of the same data type as that of the array name (see Section 6.2.3.3). You can define an array element or an entire array before program execution with a DATA statement. During program execution, you can define an array element with an assignment or input statement; and an entire array with an input statement.

6.2.3.1 Array Declarators

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

An array declarator has the form:

a(d[,d] ...)

where:

a

is the symbolic name of the array, that is, the array name. (Section 5.2.3 gives the form of a symbolic name.)

d

is a dimension declarator; d can specify both a lower bound and an upper bound as follows:

[dl:]du

where:

dl

is the lower bound of the dimension.

du

is the upper bound of the dimension. (An asterisk (*) can also occur as an upper bound, but only of the last dimension.)

The number of dimension declarators indicates the number of dimensions in the array. The number of dimensions can range from one to seven.

The value of the lower-bound dimension declarator can be negative, zero, or positive. The value of the upper-bound dimension declarator must be greater than or equal to that of the corresponding lower-bound dimension declarator. The number of elements in the dimension is $du - dl + 1$. If a lower bound is not specified, it is assumed to be one, and the value of the upper bound specifies the number of elements in that dimension. For example, a dimension declarator of 50 indicates that the dimension contains 50 elements. The upper bound in the last dimension declarator in a list of dimension declarators may be an asterisk (*); an asterisk marks the declarator as an assumed-size array declarator (see Section 10.1.1.2).

Each dimension bound is an integer arithmetic expression in which each operand is a constant, a dummy argument, or a variable in a common block. The expression is converted to an integer if necessary.

The type of a variable used in a bound expression cannot be changed by a later type declaration.

NOTE

Array references and references to user-defined functions should not be used in dimension bounds expressions.

Dimension bounds that are not constant expressions can be used in a subprogram to define adjustable arrays. You can use adjustable arrays within a single subprogram to process arrays with different dimension bounds by specifying the array name as a subprogram argument, and by either specifying the bounds as subprogram arguments or by placing the bounds in a common block. See Section 10.1.1.1 for more information on adjustable arrays. Dimension bounds that are not constant expressions are not permitted in a main program.

The number of elements in an array is equal to the product of the number of elements in each dimension.

An array name can appear in only one array declarator within a program unit.

6.2.3.2 Array Subscripts

A subscript qualifies an array name. A subscript is a list of expressions, called subscript expressions, enclosed in parentheses, that determine which element in the array is referred to. The subscript is appended to the array name it qualifies.

NOTE

The generic term *scalar reference* is used throughout this manual to refer collectively to all references that resolve to single, typed data items. Subscripted array references fall into the scalar reference category.

Section 6.2.6 provides a thorough discussion of this new terminology.

A subscript has the form:

(s[,s]...)

where:

s

is a subscript expression.

A subscripted array reference must contain one subscript expression for each dimension defined for that array (one for each dimension declarator).

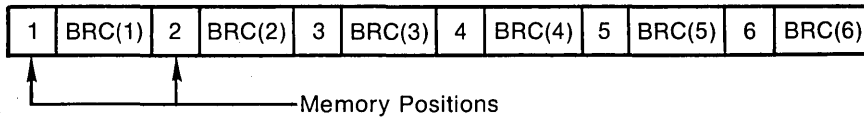
Each subscript can be any valid arithmetic expression. However, noninteger subscript expressions are converted to integers before use (any fractional parts are truncated).

6.2.3.3 Arrangement of Array Elements in Storage

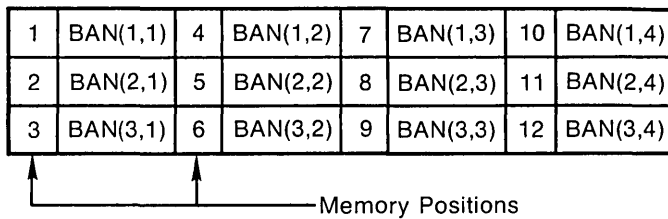
As discussed earlier in this section, you can think of the dimensions of an array as rows, columns, and levels or planes. However, FORTRAN always stores an array in memory as a linear sequence of values. A one-dimensional array is stored with its first element in the

first storage location and its last element in the last storage location of the sequence. A multidimensional array is stored so that the leftmost subscripts vary most rapidly. This is called the “order of subscript progression.” For example, Figure 6-1 shows array storage in one, two, and three dimensions.

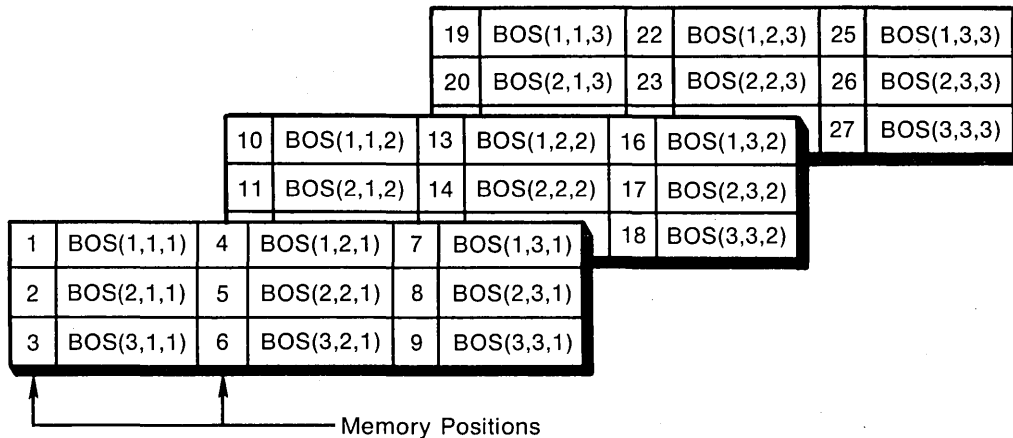
One-Dimensional Array BRC (6)



Two-Dimensional Array BAN (3,4)



Three-Dimensional Array BOS (3,3,3)



ZK-616-82

Figure 6-1: Array Storage

6.2.3.4 Data Type of an Array

The data type of an array is specified in the same way as the data type of a variable; that is, the data type of an array is specified implicitly by the initial letter of the name or explicitly by a type declaration statement.

All the values in an array have the same data type. Any value assigned to an array element is converted to the data type of the array. If an array is named in a **DOUBLE PRECISION** statement, for example, the compiler allocates an 8-byte storage location for each element of the array. When a value of any type is assigned to any element of that array, the value is converted to double precision.

6.2.3.5 Array References Without Subscripts

In the following statements, you can specify an array name without a subscript to indicate that the entire array is to be used (or defined):

- **COMMON** statement
- **DATA** statement
- **EQUIVALENCE** statement
- **NAMelist** statement
- **SAVE** statement
- **I/O** statements
- Type declaration statements

You can also use unsubscripted array names as dummy arguments in **FUNCTION**, **SUBROUTINE**, and **ENTRY** statements, and as actual arguments in references to external procedures. The use of unsubscripted array names is not permitted in all other types of statements.

6.2.3.6 Adjustable Arrays

Adjustable arrays allow subprograms to manipulate arrays of variable dimensions. To use an adjustable array in a subprogram, you specify the array bounds, as well as the array's name, as subprogram arguments. The bounds may also be given in a common block. See Section 10.1.1.1 for more information.

6.2.3.7 Assumed-Size Arrays

Assumed-size arrays are similar to adjustable arrays. With assumed-size arrays, however, an asterisk is used to specify the upper bound of the last dimension. Section 10.1.1.2 describes the rules governing the dimensions that are assumed.

6.2.4 Character Substrings

A character substring is a contiguous segment of a character variable or character array element.

A character substring reference has one of the following forms:

`v([e1]:[e2])`

`a(s[,s]...) ([e1]:[e2])`

where:

v

is a character variable name.

a

is a character array name.

s

is a subscript expression.

e1

is a numeric expression that specifies the leftmost character position of the substring.

e2

is a numeric expression that specifies the rightmost character position of the substring.

Character positions within a character variable or array element are numbered from left to right, beginning at one. For example, `LABEL(2:7)` specifies the substring beginning with the second character position and ending with the seventh character position of the character variable `LABEL`. If the `CHARACTER*8` variable `LABEL` has a value of `XVERSUSY`, then the substring `LABEL(2:7)` has a value of `VERSUS`.

If the value of the numeric expression `e1` or `e2` is not of type integer, it is converted to an integer value by truncating any fractional part before use.

The values of the numeric expressions `e1` and `e2` must meet the following conditions:

`1 .LE. e1 .LE. e2 .LE. len`

where:

len

is the length of the character variable or array element.

If `e1` is omitted, FORTRAN assumes that `e1` equals one. If `e2` is omitted, FORTRAN assumes that `e2` equals `len`.

For example, `NAMES(1,3)(:7)` specifies the substring starting with the first character position and ending with the seventh character position of the character array element `NAMES(1,3)`.

6.2.5 Records

The VAX FORTRAN record-handling capability is an extension to the FORTRAN-77 standard. It enables you to declare and operate on multifield records in your FORTRAN programs. It also enables you to access records in the VAX Common Data Dictionary (CDD) for use in your programs.

NOTE

A VAX FORTRAN record should not be confused with an RMS I/O record. A VAX FORTRAN record is a named data entity, consisting of one or more fields, that you create in your program.

6.2.5.1 Overview of Records and Structures

A record is a composite, or aggregate, entity containing one or more record elements, or fields. In this respect, it is similar to an array. It differs from an array, however, in the following respects:

- Unlike arrays, which are defined by means of a single declaration statement, creating a record is a multistep process. Creating a record requires:
 - A multistatement declaration, called a structure declaration, in which the form of the record is defined.
 - A RECORD statement that establishes the referenced structure as a record in memory, that is, as a named data entity. More than one RECORD statement can refer to a given structure declaration.
- Unlike arrays, whose data elements must be of the same data type, records allow you to organize heterogeneous data elements within one structure and to operate on them either individually or collectively. Because they can be composed of heterogeneous data elements, records are not typed as arrays are.
- Unlike arrays, each element of a record can be—and usually is—named. References to a record element consist of the name of the record and the name of the desired element.

You define the form of a record with a group of statements called a structure declaration block.

You establish a structure declaration in memory by specifying the name of the structure declaration in a RECORD statement.

A structure declaration block can include one or more of the following items:

- *Typed data declarations (variables or arrays)*: Typed data declarations in structure declarations have the form of normal FORTRAN typed data declarations. Data items with different types can be freely intermixed within a structure declaration; for example, INTEGER and LOGICAL data items can be declared in the same structure.
- *Substructure declarations*: Substructures can be established with a structure by means of either a nested structure declaration or a RECORD statement.
 - Structure declarations can be nested within structure declarations. A nested structure declaration must have one or more field names specified on its STRUCTURE statement. A nested structure declaration can optionally be given a structure name for later reference by a RECORD statement.
 - The fields in another, previously declared, structure declaration can be incorporated in a structure by including a RECORD statement, naming the other structure, within a structure declaration. This feature enables you to create a structure declaration and then include it, as necessary, as a substructure declaration within other structure declarations. Depending on the needs of an application, this can have advantages over the use of nested structure declarations, which are individually coded within a containing, outer structure.
- *Mapped field declarations*: Mapped field declarations are made up of one or more typed data declarations, substructure declarations (structure declarations and RECORD statements), or other mapped field declarations.

Mapped field declarations are defined by a block of statements called a union declaration. Unlike typed data declarations, all mapped field declarations that are made within a single union declaration share a common location within the containing structure. This capability is analogous to the use of EQUIVALENCE statements to give names to variables. In other languages, it is called a “variant record” capability.

- *Unnamed fields*: Unnamed fields can be declared in a structure by specifying the pseudo-name %FILL in place of an actual field name. You can use this mechanism to generate empty space in a record for purposes such as alignment.

For a detailed description of the syntax and use of RECORD statements and structure declarations, see Sections 8.13 and 8.15, respectively.

6.2.5.2 Arrangement of Records in Storage

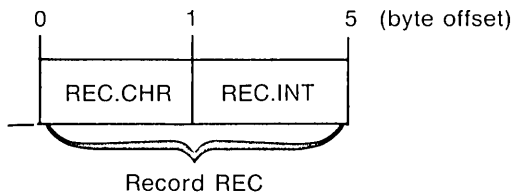
FORTRAN stores a record in memory as a linear sequence of values, with the record’s first element in the first storage location and its last element in the last storage location. No gaps are left between elements. A record array is stored in a similar fashion, with no gaps between array elements.

The following example contains a structure declaration and record statement, and shows how the resulting records are stored in memory.

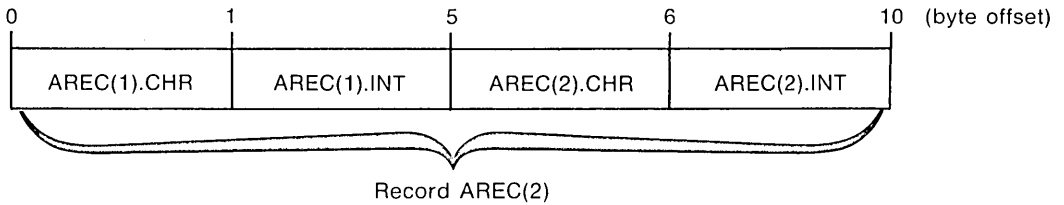
Source Code:

```
STRUCTURE /STRA/  
  CHARACTER*1 CHR  
  INTEGER*4 INT  
END STRUCTURE  
:  
:  
:  
RECORD /STRA/ REC,AREC(2)
```

Memory Diagram:



ZK-1844-84



ZK-1843-84

The next example is similar but involves a record that is more complex than the records in the preceding example. The record in this example includes a substructure.

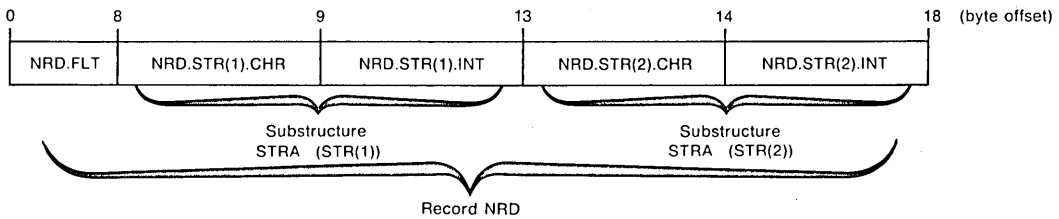
Source Code:

```

STRUCTURE /STRB/
  REAL*8 FLT
  RECORD /STRA/ STR(2)
END STRUCTURE
.
.
RECORD /STRB/ NRD

```

Memory Diagram:



ZK-1842-84

Unions cause the storage of the associated mapped fields to be overlaid, as the following example illustrates:

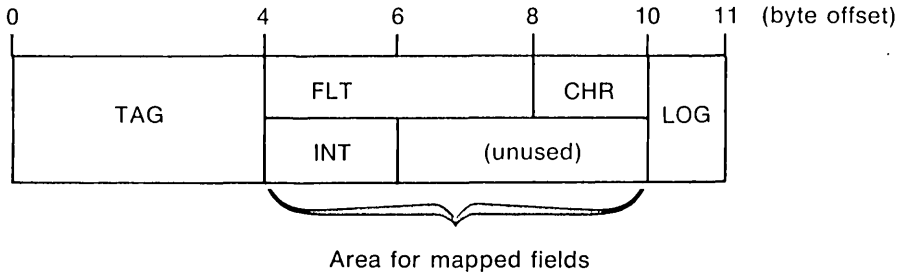
Source Code:

```

STRUCTURE /STR/
  INTEGER*4 TAG
  UNION
  MAP
    REAL*4 FLT
    CHARACTER*2 CHR
  END MAP
  MAP
    INTEGER*2 INT
  END MAP
  END UNION
  LOGICAL*1 LOG
END STRUCTURE
.
.
RECORD /STR/REC

```

Memory Diagram:



ZK-1845-84

The next section describes how to refer to records and to fields within records.

6.2.5.3 Record and Field References

Fields are the atomic units of records; they correspond to substructures or to ordinary variables or array elements. Fields within a record can be operated on collectively (that is, as part of an operation involving an entire record) or individually.

There are two forms of record references: aggregate field and scalar field references.

- An aggregate field reference refers to a composite, or structured, data item, that is, a record structure or a record substructure.
- A scalar field reference refers to a typed data item, that is, a variable or an array element.

Aggregate Field Reference:

record-name[.aggregate-field-name....aggregate-field-name]

Scalar Field Reference:

record-name.aggregate-field-name[....aggregate-field-name].scalar-field-name

where:

record-name

The name used in a RECORD statement to identify a record. See Section 8.13 for a description of the RECORD statement.

aggregate-field-name

The name of a field that is a substructure (that is, a record or a nested structure declaration) within the record structure identified by the record name.

See Section 8.15 for a description of how fields are specified within structure declarations.

scalar-field-name

The name of a typed data item defined within a structure declaration.

NOTE

The generic term *scalar reference* is used throughout this manual to refer collectively to all references that resolve to single, typed data items. Scalar field references fall into the scalar reference category.

The generic term *aggregate reference* is used throughout this manual to refer collectively to all references that resolve to references to structured data items, that is, records and nested structure declarations. Aggregate field references are the only references that fall into this category.

Section 6.2.6 provides a thorough discussion of this new terminology.

The following considerations and restrictions apply to the various forms of record references:

- **Aggregate Field References**

An aggregate field reference consists of the name of a record (as specified in a RECORD statement) and zero or more levels of aggregate field names.

Aggregate record assignments are permitted; that is, aggregate field references can be specified on the left-hand side of an assignment statement.

- **Scalar Field References**

A scalar field reference consists of the name of a record (as specified in a RECORD statement) and zero or more levels of aggregate field names followed by the name of a scalar field.

A scalar field reference refers to a single, typed data item and can be treated like a normal reference to a FORTRAN variable or array element. Scalar field references can be used in statement functions and in executable statements. They cannot, however, be used in COMMON, SAVE, NAMELIST, or EQUIVALENCE statements.

Type conversion rules for scalar field references are the same as those for variables and array elements.

- **Records in I/O Statements**

Aggregate field references can be used in unformatted I/O statements (one I/O record is written no matter how many aggregate and array name references appear in the I/O list), but cannot be used in formatted and NAMELIST I/O statements.

- **Records as Arguments**

Aggregate field references can be used as both dummy and actual arguments. The declaration of the dummy record in the subprogram must match the form of the record declared in the calling program unit, that is, each structure must have the same number and types of fields in the same order. The ordering of map fields within a union declaration is irrelevant.

Records are passed by reference. Aggregate field references are treated like normal variables. Adjustable arrays are allowed in RECORD statements used as dummy arguments.

NOTE

Because periods are used in record references to separate fields, you should not use relational operators (for example, .EQ., .XOR.) logical constants (.TRUE. or .FALSE.) and logical expressions (.AND., .NOT., .OR.) as field names in structure declarations.

Examples

The examples of record and field references shown here are based on the record structure APPOINTMENT (described at length in Section 8.15.1) and the following RECORD statement:

```
RECORD /APPOINTMENT/ NEXT_APP,APP_LIST(10)
```

The preceding statement results in the creation of both a variable named NEXT__APP and a 10-element array named APP__LIST. Both the variable and each element of the array have the form of the structure APPOINTMENT.

The declarations of the record structure APPOINTMENT and its substructure DATE are repeated here to show the fields used in the field references in the examples that follow.

Structure DATE:

```
STRUCTURE /DATE/  
    LOGICAL*1 DAY, MONTH  
    INTEGER*2 YEAR  
END STRUCTURE
```

Structure APPOINTMENT:

```
STRUCTURE /APPOINTMENT/  
    RECORD /DATE/ APP_DATE  
    STRUCTURE /TIME/ APP_TIME (2)  
        LOGICAL*1 HOUR, MINUTE  
    END STRUCTURE  
    CHARACTER*20 APP_MEMO (4)  
    LOGICAL*1 APP_FLAG  
END STRUCTURE
```

Each of the following examples of record and field references is introduced by a brief description.

Aggregate Field References:

- The record NEXT_APP:
NEXT_APP
- The field APP_TIME(1), an array field of the record NEXT_APP:
NEXT_APP.APP_TIME(1)
- The field APP_DATE, a 4-byte array field in the record array APP_LIST(3):
APP_LIST(3).APP_DATE

Scalar Field References:

- The field APP_FLAG, a LOGICAL field of the record NEXT_APP:
NEXT_APP.APP_FLAG
- The field HOUR, a LOGICAL*1 subfield of field APP_TIME(1) of record NEXT_APP:
NEXT_APP.APP_TIME(1).HOUR

- The first character of APP_MEMO(1), a CHARACTER*20 field of the record NEXT_APP:

```
NEXT_APP,APP_FLAG(20)(1:1)
```

- The field MONTH, a LOGICAL*1 subfield of field APP_DATE of record array APP_LIST(1):

```
APP_LIST(1),APP_DATE.MONTH
```

6.2.6 Terminology Used to Refer to Data Items

Constants, variables, arrays, scalar fields, aggregate fields, character substrings, and expressions can be specified in many places in a FORTRAN program. FORTRAN statements and expressions have individual restrictions governing which of these items can be used in them and in what form. Thus, to avoid repeatedly enumerating lists of the various items that can be specified with the various statements and expressions, the items are divided into four general categories. The names of these categories are used throughout this manual to identify what can be included in a particular statement or expression. The categories are as follows:

- *Scalar Reference*—resolves itself to a reference to a single, typed data item: a variable, array element, constant, character substring, or expression.
- *Scalar Memory Reference*—same as scalar reference, excluding constants and expressions.
- *Array Name Reference*—resolves itself to a reference to an array.
- *Aggregate Reference*—resolves itself to a reference to a structured data item.

References formed from the following data declarations can be used to illustrate the three types of reference:

```

INTEGER INT, INTARY (10)
.
.
STRUCTURE /STRA/
  INTEGER INTFLD, INTFLDARY (10)
END STRUCTURE
.
.
STRUCTURE /STRB/
  CHARACTER*20 CHARFLD
  INTEGER INTFLD, INTFLDARY (10)
  STRUCTURE STRUCFLD
    COMPLEX CPXFLD, CPXFLDARY (10)
  END STRUCTURE
  RECORD /STRA/ RECFLD, RECFLDARY (10)
END STRUCTURE
.
.
RECORD /STRB/ REC, RECARY (10)

```

Examples of references, by category, that can be formed from the preceding data declarations are as follows:

- **Scalar References:**

INT	REC.CHARFLD(5:10)
INTARY(1)	RECARY(1).CHARFLD(5:10)
REC.INTFLD	RECARY(1).INTFLD
REC.INTFLDARY(1)	RECARY(1).INTFLDARY(1)
REC.RECFLD.INTFLD	RECARY(1).RECFLD.INTFLD
REC.STRUCFLD.CPXFLD	RECARY(1).STRUCFLD.CPXFLD
REC.RECFLD.INTFLDARY(1)	RECARY(1).RECFLD.INTFLDARY(1)
REC.RECFLDRAY(1).INTFLD	RECARY(1).RECFLDRAY(1).INTFLD
REC.RECFLDRAY(1).INTFLDARY(1)	RECARY(1).RECFLDRAY(1).INTFLDARY(1)
REC.CHARFLD	

- **Array Name References:**

INTARY	REC.STRUCFLD.CPXFLDARY
RECARY	RECARY(1).INTFLDARY
REC.INTFLDARY	RECARY(1).RECFLDARY
REC.RECFLDARY	RECARY(1).RECFLD.INTFLDARY
REC.RECFLD.INTFLDARY	RECARY(1).STRUCFLD.CPXFLDARY
REC.RECFLDARY(1).INTFLDARY	RECARY(1).RECFLDARY(1).INTFLDARY

- **Aggregate References:**

REC	REC.RECFLDARY(1)
RECARY(1)	RECARY(1).RECFLD
REC.RECFLD	RECARY(1).STRUCFLD
REC.STRUCFLD	RECARY(1).RECFLDARY(1)

Scalar reference, array name reference, and aggregate reference are used throughout this manual to indicate where these various categories of data items can be specified. Note that constants are included in the scalar reference category.

6.3 Expressions

An expression represents a single value. An expression can consist of a scalar field reference or function reference; or combinations of these references plus certain other elements, called operators. Operators specify computations to be performed on the values of the data items and a single result is obtained.

Expressions are classified as arithmetic, character, relational, or logical. Arithmetic expressions produce numeric values; character expressions produce character values; and relational and logical expressions produce logical values.

6.3.1 Arithmetic Expressions

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

An arithmetic element can be any of the following:

- A numeric scalar reference
- An arithmetic expression enclosed in parentheses
- A numeric function reference

The term “numeric,” as used above, includes logical data, because logical data is treated as integer data when used in an arithmetic context.

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

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition or unary plus
-	Subtraction or unary minus

These operators are called binary operators because each is used with two elements. The plus (+) and minus (-) symbols are also unary operators: when they immediately precede an arithmetic element and are not immediately preceded by an arithmetic element, they denote a positive or negative value.

A variable or array element must have a defined value before it can be used in an arithmetic expression.

Any arithmetic operation whose result is not mathematically defined is not allowed. Examples of this are dividing by zero and raising a zero-valued base to a zero-valued or negative-valued power. Raising a negative-valued base to a real power is also not allowed.

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

Operator	Precedence
**	First
* and /	Second
+ and -	Third

When two or more operators of equal precedence (such as + and -) appear, they can be evaluated in any order, as long as the order of evaluation is algebraically equivalent to a left-to-right order of evaluation. Exponentiation, however, is evaluated from right to left. For example, $A**B**C$ is evaluated as $A ** (B**C)$; $B**C$ is evaluated first, and then A is raised to the resulting power.

Normally, two operators cannot be placed in succession. When the second operator is unary (+ or -), as an extension to the ANSI standard, VAX FORTRAN allows two consecutive operators within an expression. This extension can be illustrated by the following two expressions:

1. $A ** B * C$
2. $A ** -B * C$

In the first example, the ** operator is evaluated first because it takes precedence over the * operator. In the second example, the * operator is evaluated first. Normally, the ** operator would be evaluated first, but because VAX FORTRAN allows the combination of the ** and - operators, the order of evaluation is affected. With the ** and - operators combined, the ** operator cannot be evaluated until after the - operator. As a result, then, the * operator is evaluated first in compliance with normal rules of precedence.

6.3.1.1 Use of Parentheses

You can use parentheses to force a particular order of evaluation. When part of an expression is enclosed in parentheses, that part is evaluated first, and the resulting value is used in the evaluation of the remainder of the expression. In the following examples, the numbers below the operators indicate the order of the evaluations:

$$4 + 3 * 2 - 6 / 2 = 7$$

$$\begin{array}{cccc} \uparrow & \uparrow & \uparrow & \uparrow \\ 2 & 1 & 4 & 3 \end{array}$$

$$(4+3) * 2 - 6 / 2 = 11$$

$$\begin{array}{cccc} \uparrow & \uparrow & \uparrow & \uparrow \\ 1 & 2 & 4 & 3 \end{array}$$

$$(4 + 3 * 2 - 6) / 2 = 2$$

$$\begin{array}{cccc} \uparrow & \uparrow & \uparrow & \uparrow \\ 2 & 1 & 3 & 4 \end{array}$$

$$((4+3) * 2 - 6) / 2 = 4$$

$$\begin{array}{cccc} \uparrow & \uparrow & \uparrow & \uparrow \\ 1 & 2 & 3 & 4 \end{array}$$

As shown in the third and fourth examples above, expressions within parentheses are evaluated according to the normal order of precedence, unless you override the order by using parentheses within parentheses.

Nonessential parentheses, as in the following expression, do not affect expression evaluation:

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

The use of parentheses to specify the evaluation order is often important in high-accuracy numerical computations. In such computations, evaluation orders that are algebraically equivalent might not be computationally equivalent when processed by a computer.

6.3.1.2 Data Type of an Arithmetic Expression

If every element in an arithmetic expression is of the same data type, the value produced by the expression is also of that data type. If elements of different data types are combined in an expression, the evaluation of that expression and the data type of the resulting value depend on a rank associated with each data type. The rank assigned to each data type is as follows:

Data Type	Rank
Logical	1 (Lowest)
INTEGER*2	2
INTEGER*4	3
REAL*4 (REAL)	4
REAL*8 (DOUBLE PRECISION)	5
REAL*16	6
COMPLEX*8 (COMPLEX)	7
COMPLEX*16 (DOUBLE COMPLEX)	8 (Highest)

The data type of the value produced by an operation on two arithmetic elements of different data types is the data type of the highest-ranked element in the operation. For example, the data type of the value resulting from an operation on an integer and a real element is real. However, an operation involving a COMPLEX*8 data type and either a REAL*8 or REAL*16 data type produces a COMPLEX*16 result.

The data type of an expression is the data type of the result of the last operation in that expression. The data type of an expression is determined according to the following conventions:

- **Integer operations**—Integer operations are performed only on integer elements. (Logical entities used in an arithmetic context are treated as integers.) In integer arithmetic, any fraction that can result from division is truncated, not rounded. For example:

$$1/4 + 1/4 + 1/4 + 1/4$$

The value of this expression is 0, not 1.

- **Real operations**—Real operations are performed only on real elements or combinations of real, integer, and logical elements. Any integer elements present are converted to the real data type by giving each a fractional part equal to zero. The expression is then evaluated using real arithmetic. Note, however, that in the statement $Y = (I/J)*X$, an integer division operation is performed on I and J, and a real multiplication is performed on that result and X.
- **REAL*8 and REAL*16 operations**—Any element in an operation in which there is a higher-precision element is converted to the data type of the higher-precision element by making the existing element the most significant portion of the higher-precision data. The least significant portion of the binary representation is zero. The expression is then evaluated in the higher-precision arithmetic.
- **Converting a real element to a higher-precision element does not increase its accuracy.** For example, a REAL variable having the value
0.3333333
is converted to (approximately)
0.3333333134651184D0
not to either
0.3333333000000000D0
or
0.3333333333333333D0
- **Complex operations**—In an operation that contains any complex element, integer elements are converted to the real data type, as previously described. The REAL or REAL*8 element thus obtained is then designated as the real part of a complex number; the imaginary part is assigned a value of zero. The expression is then evaluated using complex arithmetic and the resulting value is of a complex data type. Operations involving COMPLEX*8 and REAL*8 elements are done as COMPLEX*16 operations; that is, the REAL*8 element is not rounded.
- **When a constant defined by a PARAMETER statement is used in an arithmetic expression, it is treated in some cases as a lower-order type even if it was explicitly typed.** For example, an INTEGER*4 constant could be treated as an constant.

These rules also generally apply to arithmetic operations in which one of the operands is a constant. However, if a real or complex constant is used in a higher-precision expression, additional precision will be retained for the constant. The effect is as if a REAL*8 or REAL*16 representation of the constant had been given. For example, the expression

```
1.0D0 + 0.3333333
```

is treated as if it were

```
1.0D0 + 0.3333333000000000D0
```

6.3.2 Character Expressions

Character expressions consist of character elements and character operators. The evaluation of a character expression yields a single value of character data type.

A character element can be any one of the following:

- A character scalar reference
- A character substring
- A character expression, optionally enclosed in parentheses
- A character function reference

The only character operator is the concatenation operator (//).

A character expression has the form:

```
character element [//character element]...
```

The value of a character expression is a character string formed by successive left-to-right concatenations of the values of the elements of the character expression. The length of a character expression is the sum of the lengths of the character elements. For example, the value of the character expression 'AB'// 'CDE' is 'ABCDE', which has a length of five.

Parentheses do not affect the value of a character expression. For example, the following character expressions are equivalent:

```
('ABC'// 'DE')// 'F'  
'ABC'// ('DE'// 'F')  
'ABC'// 'DE'// 'F'
```

Each of these character expressions has the value 'ABCDEF'.

If a character element in a character expression contains spaces, the spaces are included in the value of the character expression. For example, 'ABC'// 'D E'// 'F' has a value of 'ABC D EF'.

6.3.3 Relational Expressions

A relational expression consists of two arithmetic expressions or two character expressions separated by a relational operator. The value of the expression is either true or false, depending on whether the stated relationship holds.

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

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

The delimiting periods are a required part of each operator.

Complex expressions can be related only by the .EQ. and .NE. operators. Complex entities are equal if their corresponding real and imaginary parts are both equal.

In an arithmetic relational expression, the arithmetic expressions are first evaluated to obtain their values. These values are then compared to determine whether the relationship stated by the operator holds. For example:

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

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

Similarly, in a character relational expression, the character expressions are first evaluated to obtain their values. These values are then compared to determine whether the relationship stated by the operator holds. In character relational expressions "less than" means "precedes in the ASCII collating sequence," and "greater than" means "follows in the ASCII collating sequence." For example:

```
'AB' .LT. 'ZZZ'
```

This expression states that 'ABZZZ' is less than 'CCCCC'. Since that relationship does hold, the value of the expression is true. If the relationship stated does not hold, the value of the expression is false.

If the two character expressions in a relational expression are not the same length, the shorter one is padded on the right with spaces until the lengths are equal. For example:

```
'ABC' .EQ. 'ABC '
```

```
'AB' .LT. 'C'
```

The first relational expression has a value of true even though the lengths of the expressions are not equal, and the second has a value of true even though 'AB' is longer than 'C'.

All relational operators have the same precedence. However, arithmetic and character operators have a higher precedence than relational operators.

As in any other expression, you can use parentheses to alter the order of evaluation of the expressions in a relational expression. However, because arithmetic and character operators are evaluated before relational operators, you need not enclose the entire arithmetic or character expression in parentheses.

A relational expression can compare two numeric expressions of different data types. In this case, the value of the expression with the lower-ranked data type is converted to the higher-ranked data type before the comparison is made.

6.3.4 Logical Expressions

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

A logical element can be any of the following:

- An integer or logical scalar reference
- A relational expression
- An integer or logical expression enclosed in parentheses
- An integer or logical function reference

The logical operators are:

Operator	Example	Meaning
.AND.	A .AND. B	Logical conjunction: The expression is true if, and only if, both A and B are true.
.OR.	A .OR. B	Logical disjunction (inclusive OR): The expression is true if either A or B, or both, are true.
.NEQV.	A .NEQV. B	Logical exclusive OR: The expression is true if A and B have different logical values; but the expression is false if both elements have the same logical value.
.XOR.	A .XOR. B	Same as .NEQV.
.EQV.	A .EQV. B	Logical equivalence: The expression is true if, and only if, both A and B have the same logical value, whether true or false.
.NOT.	.NOT. A	Logical negation: The expression is true if, and only if, A is false.

The delimiting periods of logical operators are required.

When a logical operator operates on logical elements, the resulting data type is logical. When a logical operator operates on integer elements, the logical operation is carried out bit-by-bit on the corresponding bits of the internal (binary) representation of the integer elements. The resulting data type is integer. When a logical operator combines integer and logical values, the logical value is first converted to an integer value, then the operation is carried out as for two integer elements. The resulting data type is integer.

A logical expression is evaluated according to an order of precedence assigned to its operators. Some logical expressions can be evaluated before all their subexpressions are evaluated. For example, if A is `.FALSE.`, the expression `A .AND. (F(X,Y) .GT. 2.0) .AND. B` is `.FALSE.` The value of the expression can be determined by testing A without evaluating `F(X,Y)`. Under these circumstances, the function subprogram F may not be called. Thus, it is uncertain whether side-effects resulting from the call—for example, changing variables in the common block—will occur.

The following list summarizes all the operators that can appear in a logical expression, in the order in which they are evaluated:

Operator	Precedence
<code>**</code>	First (Highest)
<code>*,/</code>	Second
<code>+, -, //</code>	Third
Relational Operators	Fourth
<code>.NOT.</code>	Fifth
<code>.AND.</code>	Sixth
<code>.OR.</code>	Seventh
<code>.XOR., .EQV., .NEQV.</code>	Eighth (Lowest)

Operators of equal rank are evaluated from left to right, except for exponentiation, which is evaluated from right to left. For example:

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

The sequence in which this logical expression is evaluated is:

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

As in arithmetic expressions, you can use parentheses to alter the normal sequence of evaluation.

Two logical operators cannot appear consecutively, unless the second operator is `.NOT.`

Chapter 7

Assignment Statements

Assignment statements define the value of a data item—a variable, array element, record (structured variable), record element, or character substring. The expression on the right side of the assignment statement's equal sign is evaluated and the resulting value is assigned to the data item.

The following statements perform assignments:

- Arithmetic assignment statement
- Logical assignment statement
- Character assignment statement
- Aggregate assignment statement
- ASSIGN statement

These statements are discussed individually in the sections that follow.

7.1 Arithmetic Assignment Statement

The arithmetic assignment statement assigns the value of the expression on the right of the equal sign to the numeric scalar memory reference on the left of the equal sign.

The arithmetic assignment statement has the form:

$$v = e$$

where:

v

is a numeric scalar memory reference.

e

is an arithmetic expression.

The equal sign does not mean "is equal to," as in mathematics. It means "is replaced by."
For example:

```
COUNT = COUNT + 1
```

This statement means, "replace the current value of the integer variable COUNT with the sum of that current value and the integer constant 1."

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

The expression e must yield a value that conforms to the range requirements of v. For example, a real expression that produces a value greater than 32767 is invalid if the entity on the left of the equal sign is an INTEGER*2 variable. Significance may be lost if an INTEGER*4 value, which can exactly represent values of approximately the range -2×10^9 to $+2 \times 10^9$, is converted to REAL*4 (including the real part of a complex constant), which is accurate to only about seven digits.

If v has the same data type as that of the expression on the right, the statement assigns the value directly. If the data types are different, the value of the expression is converted to the data type of the entity on the left of the equal sign before it is assigned. Table 7-1 summarizes the data conversion rules for assignment statements.

Examples of valid and invalid assignment statements are:

Valid

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

```
PI = 3.14159
```

```
SUM = SUM + 1.
```

```
NEW = RECROD1, FIELD1
```

```
SYMBOL(I), DEFINED = , TRUE.
```

Invalid

```
3.14 = A - B
```

Entity on the left must be a numeric scalar memory reference.

```
-J = I**4
```

Entity on the left must not be signed.

```
ALPHA = ((X+6)*B*B / (X-Y)
```

Left and right parentheses do not balance.

```
ICOUNT = A / B(3:7)
```

Expressions on the right cannot be of character data type if the entity on the left is not of character data type.

Table 7-1: Conversion Rules for Assignment Statements

Variable or Array Element (V)	Expression (E)					
	Integer or Logical	REAL	REAL*8	REAL*16	COMPLEX	COMPLEX*16
Integer or Logical	Assign E to V	Truncate E to integer and assign to V	Truncate E to integer and assign to V	Truncate E to integer and assign to V	Truncate real part of E to integer and assign to V; imaginary part of E is not used	Truncate real part of E to integer and assign to V; imaginary part of E is not used
REAL	Append fraction (.0) to E and assign to V	Assign E to V	Assign MS* portion of E to V; LS* portion of E is rounded	Assign MS* portion of E to V; LS* portion of E is rounded	Assign real part of E to V; imaginary part of E is not used	Assign MS* portion of the real part of E to V; LS* portion of the real part of E is rounded; imaginary part of E is not used
REAL*8	Append fraction (.0) to E and assign to V	Assign E to MS* portion of V; LS* portion of V is 0	Assign E to V	Same as above	Assign real part of E to MS* of V; LS* portion of V is 0; imaginary part of E is not used	Assign real part of E to V; imaginary part of E is not used
REAL*16	Same as above	Same as above	Assign E to MS* portion of V; LS* portion of V is 0	Assign E to V		Assign real part of E to MS* portion of V; LS* portion of real part of V is 0. Imaginary part of E is not used
COMPLEX	Append fraction (.0) to E and assign to real part of V; imaginary part of V is 0.0	Assign E to real part of V; imaginary part of V is 0.0	Assign MS* portion of E to real part of V; LS* portion of E is rounded; imaginary part of V is 0.0	Assign MS* portion of E to real part of V; LS* portion of E is rounded; imaginary part of V is 0.0	Assign E to V	Assign MS* portion of real part of E to real part of V; LS* portion of real part of E is rounded. Assign MS* portion of imaginary part of E to imaginary part of V; LS* portion of imaginary part of E is rounded.
COMPLEX*16	Append fraction (.0) to E and assign to V; imaginary part of V is 0.0	Assign E to MS* portion of real part of V; imaginary part of V is 0.0	Assign E to real part of V; imaginary part is 0.0	Same as above	Assign real part of E to MS* portion of real part of V; LS* portion of real part is 0. Assign imaginary part of E to MS* portion of imaginary part of V; LS* portion of imaginary part is 0.	Assign E to V

*MS = most significant (high order)
 LS = least significant (low order)

7.2 Logical Assignment Statement

The logical assignment statement assigns the value of the logical expression on the right of the equal sign to the logical scalar memory reference on the left of the equal sign. See Table 7-1 for conversion rules.

The logical assignment statement has the form:

$$v = e$$

where:

v

is a logical scalar memory reference.

e

is a logical expression.

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

Examples of logical assignment statements are:

PAGEND = .FALSE.

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

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

7.3 Character Assignment Statement

The character assignment statement assigns the value of the character expression on the right of the equal sign to the character scalar memory reference on the left of the equal sign.

The character assignment statement has the form:

$$v = e$$

where:

v

is a character scalar memory reference.

e

is a character expression.

If the length of e is greater than the length of v, the character expression is truncated on the right.

If the length of e is less than the length of v, the character expression is filled on the right with spaces.

The expression e must be of character data type. You cannot assign a numeric value to a character scalar memory reference.

Note that by assigning a value to a character substring you do not affect character positions in the character scalar memory reference not included in the substring. If a character position outside of the substring has a value previously assigned, it remains unchanged; if the character position is undefined, it remains undefined.

Examples of valid and invalid character assignment statements follow. Note that all scalar memory references in the examples are assumed to be of character data type.

Valid

```
FILE = 'PROG2'
```

```
REVOL(1) = 'MAR'// 'CIA'
```

```
LOCA(3:8) = 'PLANT5' TEXT(I,J+1)(2:N-1) = NAME//X
```

Invalid

```
'ABC' = CHARS
```

Element on the left must be a character variable, array element, or substring reference.

```
CHARS = 25
```

Expression on the right must be of character data type.

```
STRING = 5HBEGIN
```

Expression on the right must be of character data type; Hollerith constants are numeric,

7.4 Aggregate Assignment Statement

The aggregate assignment statement assigns the value of each field of the aggregate on the right of the equal sign to the corresponding field of the aggregate on the left. Note that both aggregates must be declared with the same structure.

The aggregate assignment statement has the form:

$$v = e$$

where:

v

is an aggregate reference (see Section 6.2.5.3) with the same structure as the aggregate represented by e.

e

is an aggregate reference (see Section 6.2.5.3) with the same structure as the aggregate represented by v.

Examples:

```
RECORD /DATA/ TODAY, THIS_WEEK(7)
STRUCTURE /APPOINTMENT/
  ...
  RECORD /DATA/ APP_DATE
  ...
END STRUCTURE
...
RECORD /APPOINTMENT/ MEETING
GET_DATE (TODAY)
DO I = 1,7
  THIS_WEEK (I) = TODAY
  THIS_WEEK (I).DAY = TODAY.DAY + 1
END DO
MEETING.APP_DATE = TODAY
```

7.5 ASSIGN Statement

The ASSIGN statement assigns a statement label value to an integer variable. The variable can then be used as either a transfer destination in a subsequent assigned GO TO statement or a format specifier in a formatted I/O statement.

The ASSIGN statement has the form:

```
ASSIGN s TO v
```

where:

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

v
is an integer variable.

The ASSIGN statement assigns the statement number to the variable. It is similar to an arithmetic assignment statement, with one exception: the variable becomes defined for use as a statement label reference and becomes undefined as an integer variable.

The ASSIGN statement must be executed before the statement(s) in which the assigned variable is to be used. Moreover, the ASSIGN statement and the statement(s) in which the assigned variable is used must occur in the same program unit. For example:

```
ASSIGN 100 TO NUMBER
```

This statement associates the variable NUMBER with the statement label 100. Arithmetic operations on the variable, as in the following statement, then become invalid because arithmetic on label values is undefined:

```
NUMBER = NUMBER + 1
```

The next statement dissociates **NUMBER** from statement 100, assigns it an integer value 10, and returns it to its status as an integer variable:

```
NUMBER = 10
```

The variable **NUMBER** can no longer be used in an assigned **GO TO** statement.

Examples of **ASSIGN** statements are:

```
ASSIGN 10 TO NSTART  
ASSIGN 99999 TO KSTOP  
ASSIGN 250 TO ERROR
```

(Note: **ERROR** must be previously defined as an integer variable.)

Chapter 8

Specification Statements

Specification statements are nonexecutable statements that are used to allocate and initialize variables, arrays, records, and structures and to define other characteristics of the symbolic names used in the program.

The specification statements, in alphabetical order, are:

- **BLOCK DATA** statement—establishes and defines common blocks and assigns initial values to entities contained in those common blocks.
- **COMMON** statement—defines one or more contiguous areas of storage.
- **DATA** statement—assigns initial values to variables, arrays, and array elements before program execution.
- **Data type declaration statement**—explicitly defines the data type of specified symbolic names.
- **DIMENSION** statement—defines the number of dimensions in an array and the number of elements in each dimension.
- **EQUIVALENCE** statement—associates two or more entities with the same storage location.
- **EXTERNAL** statement—allows use of user-supplied procedures as arguments to subprograms. (See Appendix A for a version of the **EXTERNAL** statement that is compatible with earlier versions of FORTRAN produced by DIGITAL.)
- **IMPLICIT** statement—overrides the implied data type of symbolic names.
- **INTRINSIC** statement—allows use of FORTRAN intrinsic functions as arguments to subprograms.
- **NAMELIST** statement—specifies lists of entities whose values may be read or written in namelist-directed I/O statements and associates the list with specified group-names.
- **PARAMETER** statement—assigns a symbolic name to a constant value. (See Appendix A for a version of the **PARAMETER** statement that is compatible with earlier versions of FORTRAN produced by DIGITAL.)

- **PROGRAM** statement—assigns a symbolic name to a main program unit.
- **RECORD** statement—establishes a record with the structure defined by the block of statements in a structure declaration.
- **SAVE** statement—retains values of local variables after a return from a subprogram.
- Structure declaration block—specifies the form, or structure, of a record.
- **VOLATILE** statement—prevents optimization from being performed on specified variables, arrays, and common blocks.

The following sections detail these statements, giving their forms and showing their use.

8.1 BLOCK DATA Statement

The **BLOCK DATA** statement, followed by a series of specification statements, assigns initial values to entities in named common blocks and, at the same time, establishes and defines those blocks.

The **BLOCK DATA** statement has the form:

```
BLOCK DATA [nam]
```

where:

nam

is a symbolic name.

You can use **COMMON**, **DATA**, **DIMENSION**, **EQUIVALENCE**, **IMPLICIT**, **PARAMETER**, **RECORD**, **SAVE**, structure declaration, and type declaration statements following a **BLOCK DATA** statement.

The specification statements that follow the **BLOCK DATA** statement establish and define common blocks, assign variables, arrays, and records to these blocks, and assign initial values to the variables, arrays, and records.

A **BLOCK DATA** statement and its associated specification statements comprise a special kind of program unit. A block data program unit must not contain any executable statements. As with other types of program units, the last statement in a block data program unit is an **END** statement.

If you use a **BLOCK DATA** statement to initialize any entity in a labeled common block, you must provide a complete set of specification statements to establish the entire block, even though some of the entities in the block do not appear in a **DATA** statement. You can use the same block data program unit to define initial values for more than one common block.

You can include the name of a block data subprogram in an **EXTERNAL** statement of a different program unit to force the **VAX** Linker to search object libraries for the block data subprogram at link time.

An example of a block data program unit follows:

```
BLOCK DATA BLKDAT
INTEGER S,X
LOGICAL T,W
DOUBLE PRECISION U
DIMENSION R(3)
COMMON /AREA1/R,S,T,U /AREA2/W,X,Y
DATA R/1.0,2*2.0/, T/.FALSE./, U/0.214537D-7/, W/.TRUE./, Y/3.5/
END
```

8.2 COMMON Statement

A **COMMON** statement defines one or more contiguous areas, or blocks, of storage. **COMMON** statements also define the order in which variables, arrays, and records are stored in each common block.

A symbolic name identifies each block. However, you can omit a symbolic name for one block in a program unit. The block without a name is known as the blank common block.

The **COMMON** statement has the form:

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

where:

cb

is a symbolic name, called a common block name. **cb** can be blank. If the first **cb** is blank, you can omit the first pair of slashes.

nlist

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

Any common block name **cb** or an omitted **cb** for blank common can occur more than once in one or more **COMMON** statements in a program unit. The list **nlist** following each successive appearance of the same common block name is treated as a continuation of the list for that common block name.

A common block name can have the same name as that of a variable, array, or record. However, it cannot be the same as a function, subroutine, or entry name in the executable program.

When you declare common blocks of the same name in different program units, these blocks all share the same storage area when the program units are combined into an executable program.

Entities are assigned storage in common blocks on a one-for-one basis. Thus, the entities assigned by a **COMMON** statement in one program unit should agree with the data type of

entities placed in a common block by another program unit. For example, if one program unit contains the statement

```
COMMON CENTS
```

and another program unit contains the statements

```
INTEGER*2 MONEY  
COMMON MONEY
```

when these program units are combined into an executable program, incorrect results may occur because the 2-byte integer variable MONEY is made to correspond to the lower-addressed two bytes of the real variable CENTS.

The following program segments show a common block in a main program and a corresponding common block in a subprogram:

Main Program

```
COMMON HEAT,X /BLK1/KILO,Q  
.  
.  
.  
CALL FIGURE  
.  
.  
.
```

Subprogram

```
SUBROUTINE FIGURE  
COMMON /BLK1/LIMA,R /ALFA,BET  
.  
.  
.  
RETURN  
END
```

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

You can use array declarators in the COMMON statement to define arrays.

8.3 DATA Statement

The DATA statement assigns initial values to variables and array elements before program execution.

The DATA statement has the form:

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

where:

nlist

is a list of one or more variable names, array names, array element names, character substring names, or implied-DO lists, separated by commas. Subscript expressions

and expressions in substring references must be integer expressions containing integer constants and implied-DO variables. The form of an implied-DO list in a DATA statement is:

(dlist, i=n1,n2[,n3])

where:

dlist

is a list of one or more array element names, character substring names, or implied-DO lists, separated by commas.

i

is the name of an integer variable.

n1,n2,n3

are each an integer constant expression, except that the expression can contain implied-DO variables of other implied-DO lists that can have this implied-DO list within their ranges.

clist

is a list of constants; clist constants have one of the following forms:

$$\left\{ \begin{array}{l} c \\ n * c \end{array} \right\}$$

where:

c

is a constant or the symbolic name of a constant.

n

defines the number of times the same value is to be assigned to successive entities in the associated nlist; n is a nonzero, unsigned integer constant or the symbolic name of an integer constant.

The DATA statement assigns the constant values in each clist to the entities in the preceding nlist. Values are assigned one by one in order as they appear, from left to right. Therefore, the number of constants must correspond exactly to the number of entities in the preceding nlist.

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

The relationship of nlist items to clist items is described in the following list.

1. If both the constant value in clist and the entity in nlist have numeric data types, conversion is based on the following rules:
 - The constant value is converted, if necessary, to the data type of the variable being initialized.

- When an octal or hexadecimal constant is assigned to a variable or array element, the number of digits that can be assigned depends on the data type of the data item. If the constant contains fewer digits than the capacity of the variable or array element, the constant is extended on the left with zeros; if the constant contains more digits than can be stored, the constant is truncated on the left.
2. If the constant value in `clist` and the entity in `nlist` are both of character data type, the conversion is based on the following rules:
 - If the constant contains fewer bytes than the length of the entity, the rightmost character positions of the entity are initialized with spaces.
 - If the constant contains more bytes than the length of the entity, the character constant is truncated on the right.
 3. If the constant value in `clist` is of numeric data type and the entity in `nlist` is of character data type, the constant and the entity must conform to the following restrictions:
 - The character entity must have a length of one character.
 - The constant must be an integer, octal, or hexadecimal constant and must have a value in the range 0 through 255.
 4. If the constant value in `clist` is a Hollerith or character constant and the entity in `nlist` is a numeric variable or numeric array element, the number of characters that can be assigned depends on the data type of the data item (see Table 6-1). If the Hollerith or character constant contains fewer characters than the capacity of the variable or array element, the constant is extended on the right with spaces. If the constant contains more characters than can be stored, the constant is truncated on the right.

When the constant and the entity conform to these restrictions, the entity is initialized with the character that has the ASCII code specified by the constant. This permits a character entity to be initialized to any 8-bit ASCII code.

The first `DATA` statement in the following example assigns zero to all 10 elements of array `A` and 4 asterisks followed by 2 spaces to the character variable `STARS`. The second `DATA` statement assigns ASCII control character codes to the character variables `BELL`, `TAB`, `LF`, and `FF`.

```
INTEGER A(10)
CHARACTER BELL, TAB, LF, FF, STARS*6
DATA A, STARS /10*0, '****'/
DATA BELL, TAB, LF, FF /7,9,10,12/
```

8.4 Data Type Declaration Statements

Type declaration statements explicitly define the data type of specified symbolic names. There are two forms of type declaration statements: numeric type declarations (see Section 8.4.1) and character type declarations (see Section 8.4.2).

You can initialize data in either form of type declaration statement by placing values bounded by slashes immediately after the symbolic name of the variable or array to be initialized. The way that initial values are assigned parallels the way that initial values are assigned in DATA statements.

The following rules apply to type declaration statements:

- Type declaration statements must precede all executable statements.
- The data type of a symbolic name can be declared only once.
- A type declaration cannot change the type of a symbolic name that has been used in a context that implicitly assumes a different type.

8.4.1 Numeric Type Declaration Statements

Numeric type declaration statements have the form:

```
type v[/clist/][,v[/clist/]]...
```

where:

type

is any of the following data type specifiers: LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or DOUBLE COMPLEX. Note that BYTE and LOGICAL*1 are equivalent.

v

is the symbolic name of a constant, variable, array, statement function or function subprogram, or array declarator.

clist

is a list of constants, as in a DATA statement. (See Section 8.3.)

You can use a numeric data type declaration statement to define arrays by including array declarators (see Section 6.2.3.1) in the list.

A symbolic name can be followed by a data type length specifier of the form *s, where s is one of the acceptable lengths for the data type being declared. Such a specification overrides the length attribute that the statement implies and assigns a new length to the specified item. If you specify a data type length specifier with an array declarator, the data type length specifier goes immediately after the array name.

You can use a type declaration statement to assign initial values to variables or arrays. This is done by specifying a list of constants (clist) in the type declaration statement. The constants specified initialize only the variable or array that immediately precedes them. The constant list cannot consist of more than one element unless it is being used to initialize an array. The list of constants used to initialize an array must contain a value for every element in the array.

Examples of numeric_type declaration statements are:

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

```
INTEGER*2 I, J, K, M12*4, Q, IVEC*4(10)
REAL*8 WX1, WXZ, WX3*4, WX5, WX6*8
REAL*16 PI/3.1415900/, E/2.7200/, QARRAY(10)/5*0.0,5*1.0/
```

8.4.2 Character Type Declaration Statements

Character type declaration statements have the form:

```
CHARACTER[*len[,]] v[*len][[/clist/],[v[*len][[/clist/]]]...
```

where:

v

is the symbolic name of a constant, variable, array, statement function, or function subprogram or array declarator.

len

is an unsigned integer constant, an integer constant expression enclosed in parentheses, or an asterisk enclosed in parentheses. The value of len specifies the length of the character data elements.

clist

is a list of constants, as in a DATA statement. (See Section 8.3.)

If you use CHARACTER*len, len is the default length specification for that list. If an item in that list does not have a length specification, the item's length is len. However, if an item does have a length specification, it overrides the default length specified in CHARACTER*len.

A length specification of asterisk—for example, CHARACTER*(*)—specifies that a function name or dummy argument assumes the length specification of the corresponding function reference or actual argument (see Chapter 6). A length specification of asterisk for the symbolic name of a constant specifies that the symbolic constant name is to assume the actual length of the constant that it represents.

The length specification must be in the range 1 to 65535. If you do not specify a length, a length of one is assumed. Note that a length specification of zero is invalid. You can use a

character type declaration statement to define arrays by including array declarators (see Section 6.2.3.1) in the list. If you specify both an array declarator and a length, the array declarator goes first.

Specifying a list of constants (clist) allows you to assign initial values to variables or arrays. The constants specified initialize only the variable or array that immediately precedes them. The constant list cannot consist of more than one element unless it is being used to initialize an array. The list of constants used to initialize an array must contain a value for every element in the array.

Examples of character type declaration statements follow.

1. The following statement specifies an array NAMES comprising one hundred 32-character elements, an array SOCSEC comprising one hundred 9-character elements, and a variable NAMETY, which is 10 characters long with an initial value of 'ABCDEFGHJ':

```
CHARACTER*32 NAMES(100), SOCSEC(100)*9, NAMETY*10/'ABCDEFGHJ'/'
```

2. The following statement specifies two 8-character variables, LAST and FIRST:

```
PARAMETER (LENGTH=4)  
CHARACTER*(4+LENGTH) LAST, FIRST
```

(Note: The PARAMETER statement is described in Section 8.11.)

3. The following statement specifies an array LETTER comprising twenty-six 1-character elements and a dummy argument, BUBBLE, which has a passed length (it is defined by the calling program):

```
SUBROUTINE S1(BUBBLE)  
CHARACTER LETTER(26), BUBBLE*(*)
```

4. The following statement is invalid; the value specified for BIGCHR is too large and the length specifier for QUEST is not an integer constant expression:

```
CHARACTER*16 BIGCHR*(60000*26), QUEST*(5*INT(A))
```

8.5 DIMENSION Statement

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

The DIMENSION statement has the form:

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

where:

a(d)

is an array declarator. (See Section 6.2.3.1.)

The DIMENSION statement allocates a number of storage elements to each array named in the statement. One storage element is assigned to each array element in each dimension, and the length of each storage element is determined by the data type of the array. The total number of storage elements assigned to an array is equal to the number produced by multiplying together the number of elements in each dimension in the array declarator. For example:

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

This statement defines ARRAY as having 16 real elements of 4 bytes each and defines MATRIX as having 125 integer elements of 4 bytes each.

The VIRTUAL statement has the same form and effect as the DIMENSION statement. VAX FORTRAN supports the VIRTUAL statement in order to be compatible with PDP-11 FORTRAN.

For further information on arrays and the storage of array elements, see Section 6.2.3.

You can also use array declarators in type declaration and COMMON statements. However, in each program unit, you can use an array name in only one array declarator.

Examples of DIMENSION statements are:

```
DIMENSION BUD(12,24,10)
DIMENSION X(5,5,5), Y(4,85), Z(100)
DIMENSION MARK(4,4,4,4)
```

```
SUBROUTINE APROC(A1,A2,N1,N2,N3)
DIMENSION A1(N1:N2), A2(N3:*)
```

8.6 EQUIVALENCE Statement

The EQUIVALENCE statement partially or totally associates two or more entities in the same program unit with the same storage location.

The EQUIVALENCE statement has the form:

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

where:

nlist

is a list of variables, array elements, arrays, or character substring references, separated by commas. You must specify at least two of these entities in each list.

Each expression in a subscript or a substring reference must be an integer constant expression. Records and record fields cannot be specified in EQUIVALENCE statements.

The EQUIVALENCE statement causes all of the entities in one parenthesized list to be allocated storage beginning at the same storage location.

You can equivalence variables of different data types. If you do, multiple components of one data type can share storage with a single component of a higher-ranked data type. For example, if you make an integer variable equivalent to a complex variable, the integer variable shares storage with the real part of the complex variable.

Examples of EQUIVALENCE statements:

1. The following EQUIVALENCE statement makes the four elements of the integer array IARR occupy the same storage as that of the double-precision variable DVAR.

```
DOUBLE PRECISION DVAR
INTEGER*2 IARR(4)
EQUIVALENCE (DVAR, IARR(1))
```

2. The following EQUIVALENCE statement makes the first character of the character variables KEY and STAR share the same storage location. The character variable STAR is equivalent to the substring KEY (1:10).

```
CHARACTER KEY*16, STAR*10
EQUIVALENCE (KEY, STAR)
```

8.6.1 Making Arrays Equivalent

When you make an element of one array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between the other elements of the two arrays. Thus, if the first elements of two equal-sized arrays are made equivalent, both arrays share the same storage space. If the third element of a 7-element array is made equivalent to the first element of another array, the last five elements of the first array overlap the first five elements of the second array.

You must not attempt to use the EQUIVALENCE statement to assign the same storage location to two or more elements of the same array. You also must not attempt to assign memory locations in a way that is inconsistent with the normal linear storage of array elements. For example, you cannot make the first element of one array equivalent to the first element of another array, and then attempt to set an equivalence between the second element of the first array and the sixth element of the other array.

For example:

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

As a result of these statements, the entire array TABLE shares part of the storage space allocated to array TRIPLE. Figure 8-1 shows how these statements align the arrays.

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

Figure 8-1: Equivalence of Array Storage

Each of the following statements also aligns the two arrays as shown in Figure 8-1:

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

Similarly, you can make arrays equivalent with nonunity lower bounds. For example, an array defined as A(2:3,4) is a sequence of eight values. A reference to A(2,2) refers to the third element in the sequence. To make array A(2:3,4) share storage with array B(2:4,4), you can use the following statement:

```
EQUIVALENCE (A(3,4), B(2,4))
```

The entire array A shares part of the storage space allocated to array B. Figure 8-2 shows how these statements align the arrays.

Each of the following statements also aligns the arrays as shown in Figure 8-2:

```
EQUIVALENCE (A, B(4,1))
EQUIVALENCE (B(3,2), A(2,2))
```

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

Figure 8-2: Equivalence of Arrays with Nonunity Lower Bounds

Only in the EQUIVALENCE statement can you identify an array element with a single subscript (that is, the linear element number), even though the array was defined as a multidimensional array. For example, the following statements align the two arrays as shown in Figure 8-1:

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

8.6.2 Making Substrings Equivalent

When you make one character substring equivalent to another character substring, the EQUIVALENCE statement also sets equivalences between the other corresponding characters in the character entities.

For example:

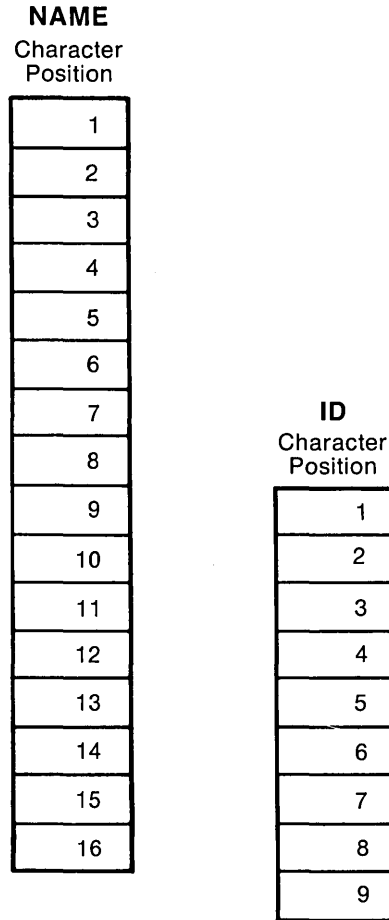
```
CHARACTER NAME*16, ID*9
EQUIVALENCE (NAME(10:13), ID(2:5))
```

As a result of these statements, the character variables NAME and ID share space as illustrated in Figure 8-3.

The following statement also aligns the arrays as shown in Figure 8-3:

```
EQUIVALENCE (NAME(9:9), ID(1:1))
```

If the character substring references are array elements, the EQUIVALENCE statement sets equivalences between the other corresponding characters in the complete arrays.



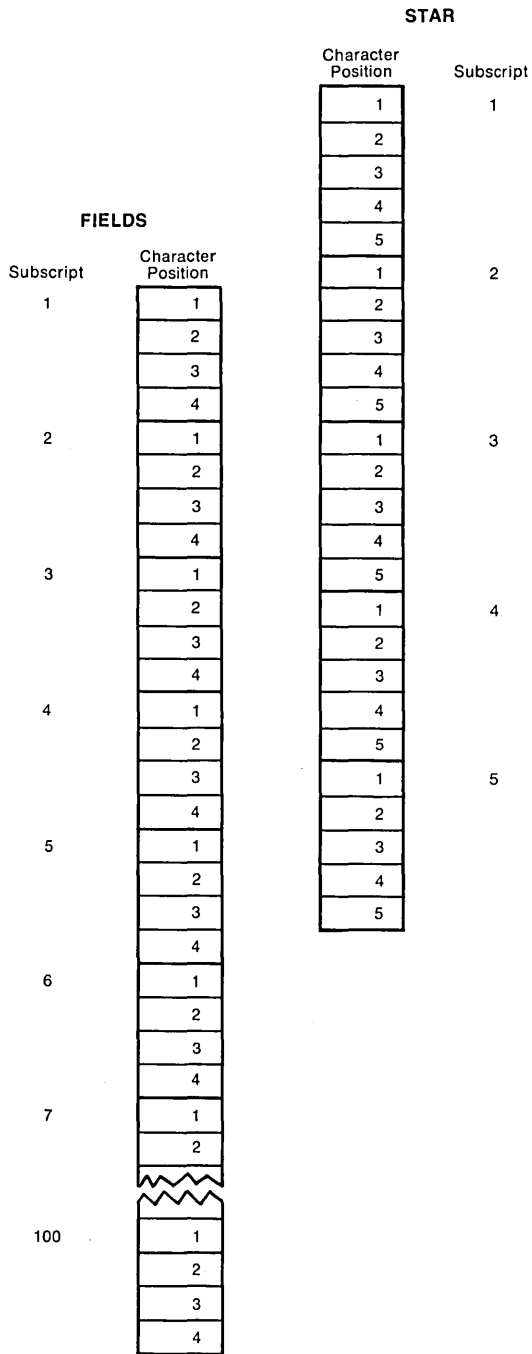
ZK-618-82

Figure 8-3: Equivalence of Substrings

Character elements of arrays can overlap at any character position. For example:

```
CHARACTER FIELDS(100)*4, STAR(5)*5  
EQUIVALENCE (FIELDS(1)(2:4), STAR(2)(3:5))
```

As a result of these statements, the character arrays FIELDS and STAR share storage space as shown in Figure 8-4.



ZK-619-82

Figure 8-4: Equivalence of Character Arrays

You cannot use the EQUIVALENCE statement to assign the same storage location to two or more substrings that start at different character positions in the same character variable or character array.

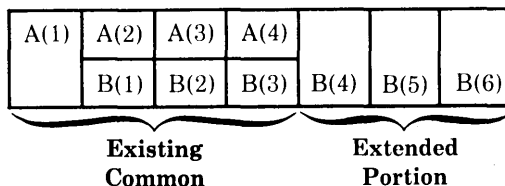
You also cannot use the EQUIVALENCE statement to assign memory locations in a way that is inconsistent with the normal linear storage of character variables and arrays.

8.6.3 EQUIVALENCE and COMMON Interaction

If you make variables or arrays equivalent to entities stored in a common block, the common block can be extended beyond its original boundaries. However, you can only extend the block beyond its last element; the extended portion cannot precede the first element in the block. The following examples show valid and invalid extensions of the common block:

Valid

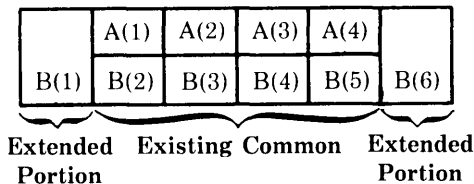
```
DIMENSION A(4), B(6)
COMMON A
EQUIVALENCE (A(2), B(1))
```



ZK-1944-84

Invalid

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



ZK-1945-84

If you assign two variables or arrays to common blocks, you cannot make them equivalent to each other.

8.7 EXTERNAL Statement

The EXTERNAL statement allows you to use the names of external procedures as arguments to other subprograms.

The subprograms mentioned in the EXTERNAL statement can never be FORTRAN intrinsic functions; they can only be user-supplied functions, subroutines, or block data subprograms. The INTRINSIC statement discussed in Section 8.9 allows intrinsic function names to be used as arguments.

The EXTERNAL statement has the form:

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

where:

v

is the symbolic name of a user-supplied subprogram or the name of a dummy argument associated with the name of a subprogram.

The EXTERNAL statement declares each symbolic name included in it to be the name of an external procedure, even if a name is the same as that of an intrinsic function. For example, if SIN is specified in an EXTERNAL statement (EXTERNAL SIN), all subsequent references to SIN are to a user-supplied function named SIN, not to the intrinsic function of the same name (see Section 10.3.3).

A name specified in an EXTERNAL statement can be used as an actual argument to a subprogram, and the subprogram can then use the corresponding dummy argument in a function reference or a CALL statement.

You can include the name of a block data subprogram in the EXTERNAL statement in order to force the VAX Linker to search the object module libraries for the block data subprogram. However, the name of the subprogram must not be used in a type declaration statement.

Note that a complete function reference used as an argument—for instance, FUNC(B) in CALL SUBR (A, FUNC(B), C)—represents a value, not a subprogram. A complete function reference is not, therefore, defined in an EXTERNAL statement.

The interpretation of the EXTERNAL statement described herein is different from that of earlier versions of FORTRAN produced by DIGITAL. See Appendix A for the earlier interpretation.

See Section 8.9 for an example of EXTERNAL statements.

8.8 IMPLICIT Statement

By default, all names beginning with the letters I through N are assumed to be of integer data type, and all names beginning with any other letter are assumed to be of REAL*4 data type. The IMPLICIT statement overrides implied data typing of symbolic names.

The IMPLICIT statement has two forms:

$$\left\{ \begin{array}{l} \text{IMPLICIT typ (a[,a]...)[,typ(a[,a]...)]...} \\ \text{NONE} \end{array} \right\}$$

where:

typ

is one of the data type specifiers. (See Chapter 6, Table 6-1.)

a

is an alphabetic specification in either of the general forms: c or c1-c2, where c is an alphabetic character. The latter form specifies a range of letters, from c1 through c2, where c1 precedes c2 in alphabetical order.

When you specify `typ` as `CHARACTER*len`, `len` specifies the length for character data type. `Len` is an unsigned integer constant or an integer constant expression enclosed in parentheses and must be in the range 1 through 65535.

The `IMPLICIT` statement assigns the specified data type to all symbolic names that begin with any specified letter, or any letter in a specified range, and that have no explicit data type declaration. For example:

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

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

Examples of `IMPLICIT` statements are:

```
IMPLICIT DOUBLE PRECISION (D)
IMPLICIT COMPLEX (S,Y), LOGICAL*1 (L,A-C)
IMPLICIT CHARACTER*32 (T-V)
IMPLICIT CHARACTER*2 (W)
```

You use an `IMPLICIT NONE` statement to override all implicit defaults. You must then explicitly declare the data types of all symbolic names in the program unit. If you specify `IMPLICIT NONE`, no other `IMPLICIT` statement can be included in the program unit.

The `IMPLICIT` statement has no effect on the default types of intrinsic functions.

By using the `/WARNINGS = DECLARATIONS` qualifier in the FORTRAN command line, you can get the benefit of `IMPLICIT NONE` (that is, be issued warnings when variables are used but not declared) without having to use the `IMPLICIT NONE` statement, a VAX FORTRAN language extension.

8.9 INTRINSIC Statement

The `INTRINSIC` statement allows you to use names of intrinsic functions as arguments to subprograms. See the appendixes for the names and descriptions of the individual FORTRAN intrinsic functions; for further information on intrinsic functions, see Chapter 10.

The `INTRINSIC` statement has the form:

```
INTRINSIC v[,v]...
```

where:

v

is the symbolic name of an intrinsic function.

The `INTRINSIC` statement declares each symbolic name included in it to be the name of an intrinsic procedure. This name can then be used as an actual argument to a subprogram, and the subprogram can then use the corresponding dummy argument in a function reference or a `CALL` statement.

8.10 NAMELIST Statement

The NAMELIST statement defines a list of variables or array names and associates that list of names with a unique group-name. The group-name is used in the namelist-directed I/O statement to identify the variables or arrays that are to be read or written.

The NAMELIST statement has the form:

```
NAMELIST /group-name/ namelist[[,] /group-name/ namelist]...
```

where:

group-name

is a symbolic name.

namelist

is a list of variable or array names, separated by commas, that is to be associated with the preceding group-name.

The namelist associates a group of entities (variables or arrays) with a single group-name, which is used by namelist-directed I/O statements in lieu of an I/O list. The unique group-name identifies a list whose entities can be modified or transferred.

The namelist entities can be of any data type and can be explicitly or implicitly typed. Array elements, character substrings, records, and record fields are not permitted in a namelist, but you can use namelist-directed I/O to assign values to elements of arrays or substrings of character variables that appear in namelists.

Only the entities specified in the namelist can be read or written in namelist-directed I/O. It is not necessary for the input records in a namelist-directed input statement to define every entity in the associated namelist.

The order of entities in the namelist controls the order in which the values are written in the namelist-directed output. Input of namelist values can be in any order.

A variable or an array name can appear in several namelists. Dummy arguments cannot appear in a namelist.

An example of a NAMELIST statement follows:

```
CHARACTER*30 NAME(25)
NAMELIST /INPUT/ NAME, GRADE, DATE, /OUTPUT/ TOTAL, NAME
```

In the preceding example, the NAMELIST statement defines two group-names: (1) INPUT with the entities NAME, GRADE, and DATE and (2) OUTPUT with the entities TOTAL and NAME.

Refer to Sections 11.4.1.3 and 11.5.1.3 for more information on namelist-directed I/O.

8.11 PARAMETER Statement

The PARAMETER statement assigns a symbolic name to a constant.

The PARAMETER statement has the form:

```
PARAMETER (p=c[,p=c]...)
```

where:

p

is a symbolic name.

c

is a constant, a compile-time constant expression, or the symbolic name of a constant.

Compile-Time Constant Expressions

A compile-time constant expression can be a compile-time logical expression, a compile-time character expression, or a compile-time arithmetic expression.

A compile-time logical expression is a logical expression in which:

- Each operand is either a constant; the symbolic name of a constant; one of the functions IAND, IOR, NOT, IEOR, ISHFT, LGE, LGT, LLE, LLT with constant operands; or another compile-time constant expression.
- Each operand has a data type of logical or integer.
- Each operator is a Boolean or relational operator.

A compile-time character expression is a character expression in which:

- Each operand is either a constant, the symbolic name of a constant, the function CHAR with a constant operand, or another compile-time constant expression.
- Each operand has a data type of character.
- Each operator is the concatenation operator //.

A compile-time arithmetic expression is an arithmetic expression in which:

- Each operand is either a constant; the symbolic name of a constant; one of the functions MIN, MAX, ABS, MOD, ICHAR, NINT, DIM, DPROD, CMPLX, CONJG, IMAG with constant operands; or another compile-time constant expression.
- Each operand has a data type of integer, real, or complex.
- Each operator is a +, -, *, /, or ** operator. (The ** operator is evaluated at compile time only if the exponent has a data type of integer.)

Symbolic Names

The data type of a symbolic name defined to be a constant is determined as follows:

- By an explicit type declaration statement preceding the defining PARAMETER statement
- By the same rules for implicit declarations that determine the data type of any other symbolic name

For example, the following PARAMETER statement is interpreted as $MU=1$, unless the PARAMETER statement is preceded by an appropriate type declaration or IMPLICIT statement, for example, REAL*8 MU:

```
PARAMETER (MU=1.23)
```

Once a symbolic name is defined to be a constant, it can appear anywhere in a program that any other constant can appear—except in FORMAT statements (where constants can only be used in variable format expressions) and as the character count for Hollerith constants. For compilation purposes, writing the name is the same as writing the value. The following rules govern the use of symbolic constant names:

- If the symbolic name is used as the length specifier in a CHARACTER declaration, it must be enclosed in parentheses. If it is used as a numeric item in a FORMAT edit description, it must be enclosed in angle brackets.
- The symbolic name of a constant cannot appear as part of another constant, it can appear as either the real or imaginary part of a complex constant.
- A symbolic name defined to be a constant can be used only within the program unit containing the defining PARAMETER statement. Also, a symbolic name can be defined only once within the same program unit.

The form and the interpretation of the PARAMETER statement described herein are different from those of the PARAMETER statement provided in earlier versions of FORTRAN produced by DIGITAL. However, VAX FORTRAN provides support for both the FORTRAN-77 and the earlier form of the PARAMETER statement; see Appendix A for information on the earlier form and interpretation.

The following sequence of statements demonstrates the use of the FORTRAN-77 PARAMETER statement:

```
REAL*4 PI, PIOV2
REAL*8 DPI, DPIOV2
LOGICAL FLAG
CHARACTER*(*) LONGNAME
PARAMETER (PI=3.1415927, DPI=3.141592653589793238D0)
PARAMETER (PIOV2=PI/2, DPIOV2=DPI/2)
PARAMETER (FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS')
```

8.12 PROGRAM Statement

The PROGRAM statement assigns a symbolic name to a main program unit.

The PROGRAM statement has the form:

```
PROGRAM nam
```

where:

nam

is the symbolic name of a source file.

The PROGRAM statement is optional. The default name for a main program unit is filename\$MAIN, where filename is the name of your source file. If filename is larger than 26 characters and a name is not specified in a PROGRAM or BLOCK DATA statement, the name is truncated to 26 characters and \$MAIN is appended to form the program name.

If you use the PROGRAM statement, it must be the first statement in the main program; however, it is the second statement in a main program that begins with an OPTIONS statement. The symbolic name must not be the name of any entity within the main program; it also must not be the same as the name of any subprogram, entry, or common block in the same executable program.

8.13 RECORD Statement

The RECORD statement creates a record of the form specified in a previously declared structure.

The effect of a RECORD statement is comparable to that of an ordinary FORTRAN type declaration except that composite, or aggregate, data items are declared instead of scalar data items.

The format of a RECORD statement is as follows:

```
RECORD /structure-name/record-namelist  
      [./structure-name/record-namelist]  
      .  
      .  
      .  
      [./structure-name/record-namelist]
```

where:

structure-name

is the name of a previously declared structure. See Section 8.15.1 for a description of structure declarations.

record-namelist

is a list of one or more variable names, array names, or array declarators, separated by commas. All of the records named in this list have the same structure and are allocated separately in memory.

Record names can be used in COMMON and DIMENSION statements. They cannot be used in DATA, EQUIVALENCE, NAMELIST, or SAVE statements.

Records initially have undefined values unless you have defined their values in structure declarations.

Examples

The following RECORD statement creates a pair of records with the form of structure declaration DATE:

```
RECORD/DATE/ TODAY , YESTERDAY
```

The preceding example creates the variables TODAY and YESTERDAY—each with the same structure—in separate areas of memory.

The following RECORD statement creates a record and an array of records with the structure CHECK.

```
RECORD /CHECK/ CURRENT_CHECK , CHECKBOOK(1000)
```

8.14 SAVE Statement

The SAVE statement causes the definition of data entities to be retained after execution of a RETURN or END statement in a subprogram.

The SAVE statement has the form:

```
SAVE [a[,a]...]
```

where:

a

is one of the following entities: a common block name (preceded and followed by a slash), a variable name, or an array name.

An entity specified by a SAVE statement within a program unit does not become undefined upon execution of a RETURN or END statement in that program unit. If the entity is in a common block, however, it may become undefined (or redefined) in another program unit.

Procedure names, blank common blocks, names of entities in a common block, and names of dummy arguments cannot be used in a SAVE statement.

A SAVE statement that does not explicitly contain a list is treated as though it contained a list of all allowable items in the program unit that contains the SAVE statement.

NOTE

It is not necessary to use `SAVE` statements in VAX FORTRAN programs. The definitions of data entities are retained automatically by VAX FORTRAN, making the use of `SAVE` statements a redundant exercise. However, its use is required by the ANSI FORTRAN Standard for programs that depend on such retention for their correct operation. If you want your programs to be transportable, you should include `SAVE` statements where your programs would otherwise require them. Note that the omission of `SAVE` statements in necessary instances is not flagged, even when you specify the `/STANDARD` qualifier on your FORTRAN command, because the compiler has no way to determine whether such dependencies exist.

8.15 Structure Declaration Block

The structure, or form, of a record is defined by a multistatement declaration. This declaration is composed of the following elements:

- *STRUCTURE statement*: This statement indicates the beginning of a structure declaration.
- *Declaration body*: The body of a structure declaration is composed of one or more field declarations. The order of the declarations determines the order of the fields within a structure.
- *END STRUCTURE statement*: This statement indicates the end of a structure declaration.

Field declarations within structure declarations consist of the following:

1. *Typed data declaration statements*. Ordinary FORTRAN type declaration; see Section 8.4. Fields can be any data type and can be dimensioned in the normal way.
2. *Substructure declarations*. A field within a structure can be a substructure composed of atomic fields and/or other substructures. There are two ways to declare substructures:
 - `RECORD` statements specifying names of other, previously declared, structure declarations to be incorporated as substructures; see Section 8.13.
 - Other, nested structure declarations, that is, one or more levels of structure declarations contained within a structure declaration.
3. *Union declarations*. A union declaration declares groups of fields that logically share a common location within a structure. Each group of fields within a union declaration is declared by a map declaration, with one or more fields per map declaration.

You use union declarations when you want to use the same area of memory to alternately contain two or more groups of fields. Whenever one of the fields declared by a union declaration is referenced in your program, that field and any other fields

in its map declaration become defined. Then, when a field in one of the other map declarations in the union declaration is referenced, the fields in that map declaration become defined, superseding the fields that were previously defined.

4. *PARAMETER statements.* A structure declaration block containing a PARAMETER statement has no effect on the meaning of the PARAMETER statement declaration. (Note: Because the PARAMETER statement is not conceptually related to the topic of structure declarations, its inclusion in a structure declaration block is not noted elsewhere in this section.)

The names specified in these statements are not the names of variables and the statements in a structure declaration do not create variables. The names are field names, and the information provided in the statements describes the layout, or form, of the structure. The ordering of both the statements and the field names within the statements is important because this ordering determines the order of the fields in records.

Sections 8.15.1 through 8.15.3 describe structure declarations, substructure declarations, and union declarations.

8.15.1 Structure Declaration

A VAX FORTRAN record comprises one or more fields. The field(s) within a VAX FORTRAN record are defined by means of a structure declaration. This declaration defines the field names, the types of data within fields, and the order and alignment of fields within a record.

Unlike type declaration statements, structure declarations do not create variables. Structured variables (referred to as “records”) are created when you use a RECORD statement containing the name of a previously declared structure. The RECORD statement can be considered as a kind of type statement. The difference is that aggregate items, rather than single items, are being defined.

The form of a structure declaration is as follows:

```
STRUCTURE [/structure-name/][field-namelist]
    field-declaration
    [field-declaration]
    .
    .
    .
    [field-declaration]
END STRUCTURE
```

where:

structure-name

The name used to identify a structure. A structure name is enclosed by slashes. If the slashes are present, a name must be specified between them.

The structure name is used in subsequent RECORD statements to refer to a structure.

Structure declarations can be nested; that is, a structure declaration can contain one or more other structure declarations. A structure name is required on the structured declaration at the outermost level of nesting. A structure name is optional for nested declarations. A nested structure declaration requires a name only when you wish to reference it elsewhere in your program in a RECORD statement.

A structure name must be unique among structure names. However, structure names can also be used to name either variables (simple or array) or fields. Thus, it is possible to have a variable named X, a structure named X, and one or more fields named X.

Structure, field, and variable names are all local to the defining program unit. When records are passed as arguments, the fields must match in type, order, and dimension.

field-namelist

A list of fields having the structure of the associated structure declaration. A field namelist is allowed only in nested structure declarations. Nested structure declarations are described in Section 8.15.2.

field-declaration

Field declarations can consist of any combination of the following types of declarations:

Substructure declarations

Union declarations

Typed data declarations

These declarations are described under the headings that follow.

Substructure declaration

A field within a structure can be a substructure composed of atomic fields and/or other substructures. See Section 8.15.2 for a description of substructure declarations.

Union declaration

A union declaration is composed of one or more mapped field declarations. The mapped fields logically share a common location within a structure. See Section 8.15.3 for a more complete description of union declarations.

Typed data declaration

The syntax of a typed data declaration within a record structure is identical to that of a normal FORTRAN type statement; that is, it includes a type (for example, INTEGER), one or more names of variables or arrays, and, optionally, one or more data initialization values.

The following rules apply to typed data declarations in record structures:

- The pseudo-name %FILL can be specified in place of a field name to create empty space in a record for purposes such as alignment. This creates an unnamed field.
- Initial values can be supplied in the field declaration statements. These initial values are supplied for all records that are declared using this structure. Fields not initialized will have undefined values when variables are declared by means of RECORD statements. Unnamed fields cannot be initialized; they are always undefined. See Sections 8.4.1 and 8.4.2 for detailed descriptions of numeric and character typed declarations.
- All field names must be explicitly typed and there are no default names. The IMPLICIT statement has no effect on statements within a structure declaration.
- All VAX FORTRAN data types are allowed in the data declarations.
- Any required array dimensions must be specified in the field declaration statements; DIMENSION statements cannot be used to define field names.
- Adjustable or assumed sized arrays and passed length CHARACTER declarations are not allowed in field declarations.
- Field names within the same declaration level must be unique, but an inner structure declaration (substructure declaration) can include field names used in an outer structure declaration without conflict.

In a structure declaration, each field offset is the sum of the lengths of the previous fields; the length of the structure therefore is the sum of the lengths of its fields. The structure is packed; you must explicitly provide any alignment that is needed by including, for example, unnamed fields of the appropriate length.

Examples

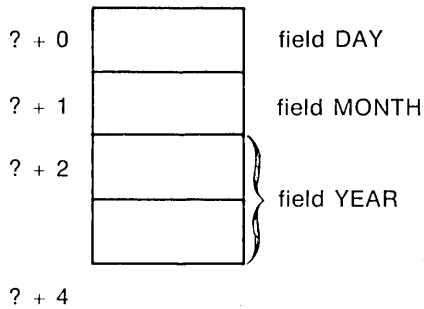
The following examples illustrate the use and form of structure declarations:

1. The structure named DATE is defined by the following declaration:

```
STRUCTURE /DATE/  
    LOGICAL*1 DAY, MONTH  
    INTEGER*2 YEAR  
END STRUCTURE
```

This structure contains three scalar fields: DAY (LOGICAL*1), MONTH (LOGICAL*1), and YEAR (INTEGER*2).

The following diagram shows the memory mapping of any record or record array element with the structure DATE.



ZK-1849-84

2. The structure named APPOINTMENT is defined by the following declaration:

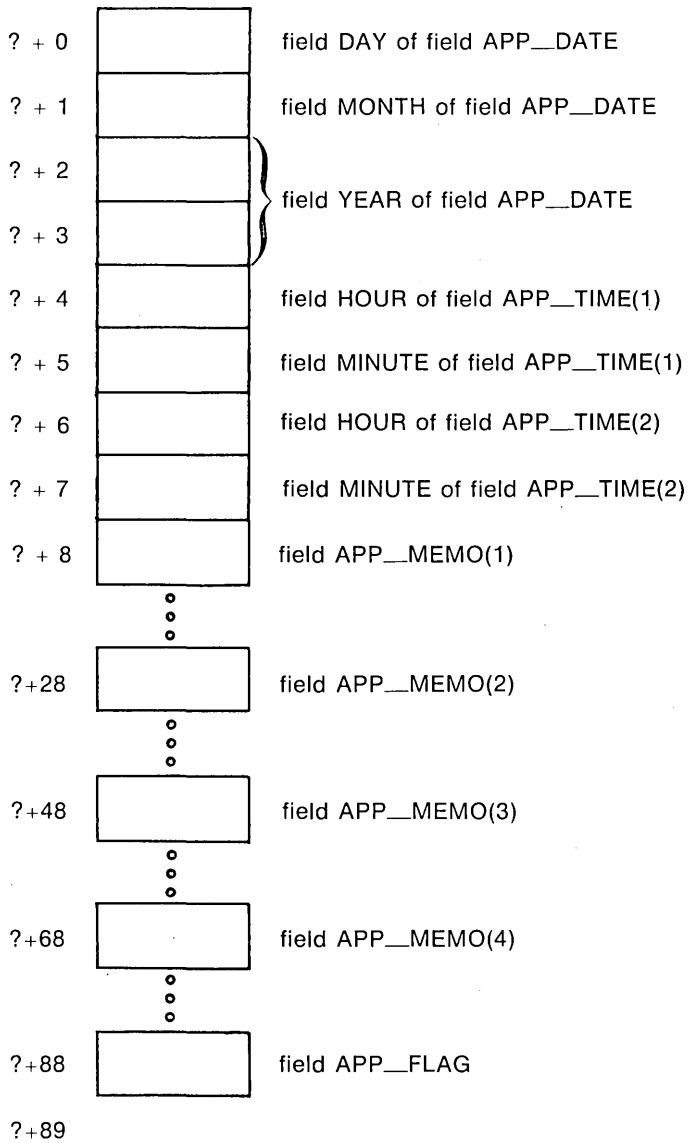
```

STRUCTURE /APPOINTMENT/
  RECORD /DATE/      APP_DATE
  STRUCTURE /TIME/   APP_TIME (2)
    LOGICAL*1        HOUR, MINUTE
  END STRUCTURE
  CHARACTER*20       APP_MEMO (4)
  LOGICAL*1          APP_FLAG
END STRUCTURE

```

APPOINTMENT contains the structure DATE (field APP_DATE) as a substructure. It also contains a substructure named TIME (field APP_TIME, an array), a CHARACTER*20 array named APP_MEMO, and a LOGICAL*1 variable named APP_FLAG. (Note: The use of substructures is described in the Section 8.15.2.)

The length of any instance of structure APPOINTMENT is 89 bytes. The following diagram shows the memory mapping of any record or record array element with the structure APPOINTMENT.



ZK-1848-84

3.15.2 Substructure Declarations

A field within a structure can itself be a structured item composed of atomic fields and/or other structured items. There are two ways of declaring substructures:

- Nested structure declaration:

A structure declaration contained within either a structure declaration or a union declaration. For obvious reasons, a structure cannot, at any level of nesting, include itself as a substructure in structure or union declarations.

One or more field names must be defined in the STRUCTURE statement for the substructure because all fields in a structure must be named and, in this case, a structure—the substructure—is being used as a field within a structure or union.

Field names within the same declaration nesting level must be unique, but an inner structure declaration can include field names used in an outer structure declaration without conflict.

The pseudo-name %FILL can be specified in place of a field name to create empty space in a record for purposes such as alignment.

- RECORD statement declaration:

A RECORD statement specifying another, previously defined, record structure to be included in the structure being declared.

See the second example in the preceding section for a sample structure declaration containing both a nested structure declaration (TIME) and an included structure (DATE).

3.15.3 Union Declarations

A union declaration is a multistatement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields.

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are two or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration.

The form of a union declaration is as follows:

```
UNION
  map-declaration
  map-declaration
  [map-declaration]
  .
  .
  .
  [map-declaration]
END UNION
```

where **map-declaration** is:

```
MAP
  field-declaration
  [field-declaration]
  .
  .
  .
  [field-declaration]
END MAP
```

where:

field-declaration

is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a typed data field within a union. See Section 8.15.1 for a more detailed description of what can be specified in field declarations.

As with normal FORTRAN type declarations, data can be initialized in field declaration statements in union declarations. Note, however, that if fields within multiple map declarations in a single union are initialized, the data declarations are initialized in the order in which the statements appear. As a result, only the final initialization takes effect and all of the preceding initializations are overwritten.

The size of the shared area established for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the field(s) declared within it.

As the variables or arrays declared in map fields in a union declaration are assigned values during program execution, the values are established in a record in the field shared with other map fields in the union. The fields of only one of the map declarations is defined within a union at any given point in the execution of a program. Note, however, if you overlay one variable with another variable that is smaller, that portion of the initial variable that is not overlaid is retained. Depending on the application, the retained portion of an overlaid variable may or may not contain meaningful data and be utilized at a later point in the program.

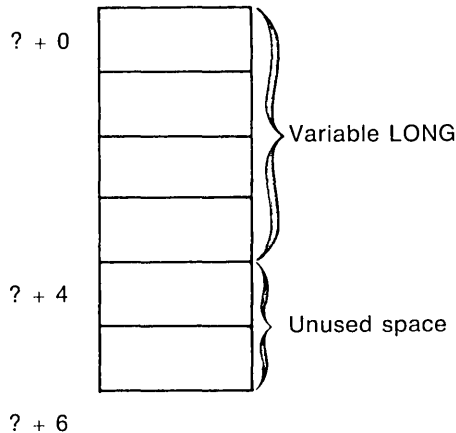
Manipulating data using union declarations is similar to what happens using EQUIVALENCE statements. The difference is that data entities specified within EQUIVALENCE statements are *concurrently* associated with a common storage location and the data residing there; whereas, union declarations enable you to use one discrete storage location to *alternately* contain a variety of fields (arrays or variables). With union declarations, only one map declaration within a union declaration can be associated at any point in time with the storage location that they share. Whenever a field within another map declaration—in the same union declaration—is referenced in your program, the fields in the prior map declaration become undefined and are succeeded by the fields in the map declaration containing the newly referenced field.

Examples

In the following example, the structure WORDS_LONG is defined. This structure contains a union declaration defining two map fields. The first map field consists of three INTEGER*2 variables (WORD_0, WORD_1, and WORD_2), and the second, an INTEGER*4 variable, LONG.

```
STRUCTURE /WORDS_LONG/  
  UNION  
    MAP  
      INTEGER*2    WORD_0, WORD_1, WORD_2  
    END MAP  
    MAP  
      INTEGER*4    LONG  
    END MAP  
  END UNION  
END STRUCTURE
```

The length of any record with the structure WORDS_LONG is six bytes. The following diagram shows the memory mapping of any record with the structure WORDS_LONG:



ZK-1846-84

8.16 VOLATILE Statement

The VOLATILE statement prevents all optimization operations from being performed on the variables, arrays, or common blocks that it identifies. As a result, for example, a variable that is declared but never referenced will still be retained if it has been declared as volatile.

VOLATILE nlist

nlist

is a list of one or more variable names, common block names, or array names, separated by commas.

If array names or common block names are used, the entire array or common block becomes volatile.

For example:

```
PROGRAM TEST
LOGICAL*1 IPI(4)
INTEGER*4 A,B,C,D,E,ILOOK
INTEGER*4 P1,P2,P3,P4
COMMON /BLK1/A,B,C
.
.
.
VOLATILE /BLK1/,D,E
EQUIVALENCE(ILOOK,IPI)
EQUIVALENCE (A,P1)
EQUIVALENCE (P1,P4)
.
.
.
```

In this example, the named common block, BLK1, and the variables D, and E are volatile. In addition, variables P1 and P4 become volatile; a direct equivalence (as in the case of P1) or indirect equivalence (as in the case of P4) causes the equivalenced variables to assume a volatile attribute.

See the *VAX FORTRAN User's Guide* for information about the optimizations performed by the VAX FORTRAN compiler and the circumstances in which you should use the VOLATILE declaration.

Chapter 9

Control Statements

Statements are normally executed in the order in which they are written. However, you can interfere with normal program flow by transferring control to another section of the program unit or to a subprogram. The transfer of control can be conditional or unconditional; that is, it can occur as a result of specified conditions being met at a certain point in the program unit or it can occur unconditionally each time a certain point is reached.

You use the FORTRAN control statements to transfer control to a point within the same program unit or to another program unit. These statements also govern iterative processing, suspension of program execution, and program termination.

The control statements and their effects are as follows:

- CALL statement—invoke a subroutine subprogram
- CONTINUE statement—transfer control to the next executable statement
- DO and DO WHILE statements—execute a block of statements repetitively
- END statement—mark the end of a program unit
- END DO statement—terminate DO and DO WHILE loops
- GO TO statement—transfer control within a program unit
- IF statement—transfer control or execute a statement (conditional)
- IF THEN, ELSE IF THEN, ELSE, and END IF statements—execute a block of statements (conditional)
- PAUSE statement—temporarily suspend program execution
- RETURN statement—return control from a subprogram to the calling program unit
- STOP statement—terminate program execution

The following sections describe these statements, in alphabetical order, giving their forms as well as examples of how they are used.

9.1 CALL Statement

The CALL statement executes a subroutine subprogram or other external procedure. It can also specify an argument list for the subroutine. (See Chapter 10 for greater detail on the definition and use of subroutines.)

The CALL statement has the form:

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

where:

sub

is the name of either (1) a subroutine subprogram or other external procedure or (2) a dummy argument associated with a subroutine subprogram or other external procedure.

a

is an actual argument. (Section 10.1 describes actual arguments.)

If you specify an argument list, the CALL statement associates the values in the list with the dummy arguments in the subroutine. It then transfers control to the first executable statement following the SUBROUTINE or ENTRY statement referenced by the CALL statement.

The arguments in the CALL statement must agree in number, order, and data type with the dummy arguments in the subroutine. They can be variables, arrays, array elements, records, record elements, record arrays, record array elements, substring references, constants, expressions, Hollerith constants, alternate return specifiers, or subprogram names. An unsubscripted array name or record array name in the argument list refers to the entire array.

Examples of CALL statements are:

```
CALL CURVE(BASE,3.14159*X,Y,LIMIT,R(LT+2))
```

```
CALL PNTOUT(A,N,'ABCD')
```

```
CALL EXIT
```

```
RECORD /GETJPI/ GETJPIARG
```

```
·  
·  
·
```

```
CALL SYS$GETJPI (,,,GETJPIARG,,)
```

```
CALL MULT(A,B,*10,*20,C)
```

The last example illustrates the use of statement label identifiers in CALL statement argument lists. The asterisks in the last CALL statement indicate that *10 and *20 are statement label identifiers. Label identifiers that are prefixed by asterisks (or ampersands (&)) are called alternate return specifiers (see Section 10.1.1.5).

9.2 CONTINUE Statement

The CONTINUE statement transfers control to the next executable statement. It is used primarily as the terminal statement of a labeled DO loop when that loop would otherwise end improperly, that is, with either a GO TO, arithmetic IF, or any other prohibited control statement.

The CONTINUE statement has the form:

```
CONTINUE
```

9.3 DO Statement

The two types of DO statements are:

- Indexed DO (DO)
- Pretested indefinite DO (DO WHILE)

DO is discussed in Section 9.3.1 and DO WHILE in Section 9.3.2.

9.3.1 Indexed DO Statement

The indexed DO statement controls iterative processing; that is, the statements in its range are repeatedly executed a specified number of times.

The DO statement has the form:

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

where:

s

is the label of an executable statement. The statement must physically follow in the same program unit. VAX FORTRAN allows the label to be omitted.

v

is a variable with an integer or real data type.

e1,e2,e3

are arithmetic expressions.

The variable *v* is the control variable; *e1*, *e2*, and *e3* are the initial, terminal, and increment parameters, respectively. If you omit the increment parameter, a default increment value of 1 is used.

The optional label that appears in the DO statement identifies the terminal statement of the DO loop. If no label appears in the DO statement, the DO loop must be terminated by the END DO statement as discussed in Section 9.4. The terminal statement must not be one of the following statements:

- Unconditional or assigned GO TO statement
- Arithmetic IF statement
- Any block IF statement
- END statement
- RETURN statement
- DO statement

The range of the DO statement includes all the statements that follow the DO statement, up to and including the terminal statement or END DO.

The DO statement first evaluates the expressions e1, e2, and e3 to determine values for the initial, terminal, and increment parameters, respectively. The increment parameter (e3) cannot be zero.

The value of the initial parameter is assigned to the control variable. If the data type of the initial, terminal, and increment parameters are not the same as the data type of the control variable, they are converted before they are used.

The number of executions of the DO range, called the iteration count, is given by:

$$[(e2 - e1 + e3)/e3]$$

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

If the iteration count is zero or negative, the body of the loop is not executed.

If the /NOF77 qualifier is specified on the FORTRAN command and the iteration count is zero or negative, the body of the loop is executed once.

9.3.1.1 DO Iteration Control

After each iteration of the DO range, the following steps are executed:

1. The value of the increment parameter (e3) is algebraically added to the control variable.
2. The iteration count (e1) is decremented.
3. If the iteration count is greater than zero, control transfers to the first executable statement after the DO statement for another iteration of the range.
4. If the iteration count is zero, execution of the DO statement terminates. The final value of the control variable is the value determined by step 1.

Note that if the data type of the control variable is real, the number of iterations of the DO range, because of rounding errors, might not be what is expected.

You can also terminate execution of a DO statement by using a statement within the range that transfers control outside the loop. The control variable of the DO statement remains defined with its current value.

When execution of a DO loop terminates and other DO loops share its terminal statement, control transfers outward to the next most enclosing DO loop in the DO nesting structure (see Section 9.3.1.2). If no other DO loop shares the terminal statement or if the DO statement is outermost, control transfers to the first executable statement after the terminal statement.

You cannot alter the value of the control variable within the range of the DO statement. However, you can use the control variable for reference as a variable within the range.

You can modify the initial, terminal, and increment parameters within the loop without affecting the iteration count.

The range of a DO statement can contain other DO statements, as long as these nested DO loops meet certain requirements. Section 9.3.1.2 describes these requirements.

You can transfer control out of a DO loop, but not into a loop from elsewhere in the program. Exceptions to this rule are described in Sections 9.3.1.3 and 9.3.1.4.

Examples of DO iteration control follow.

1. The following statement specifies 25 iterations; K=49 during the final iteration, K=51 after the loop.

```
DO 100 K=1,50,2
```

2. The following statement specifies 27 iterations; J=-2 during the final iteration, J=-4 after the loop.

```
DO 350 J=50,-2,-2
```

3. The following statement specifies 5 iterations; IVAR=5 during the final iteration, IVAR=6 after the loop.

```
DO 25 IVAR=1,5
```

4. The following statement specifies 9 iterations; NUMBER=37 during the final iteration, NUMBER=41 after the loop. The terminating statement of the DO loop must be END DO.

```
DO NUMBER=5,40,4
```

The following example illustrates how a common typing error can cause errors with DO loops; a decimal point is typed in place of a comma.

```
DO 40 M=2.10
```

9.3.1.2 Nested DO Loops

A DO loop can contain one or more complete DO loops. The range of an inner nested DO loop must lie completely within the range of the next outer loop. Nested loops can share a labeled terminal statement but not an unlabeled END DO statement.

Figure 9-1 illustrates nested loops.

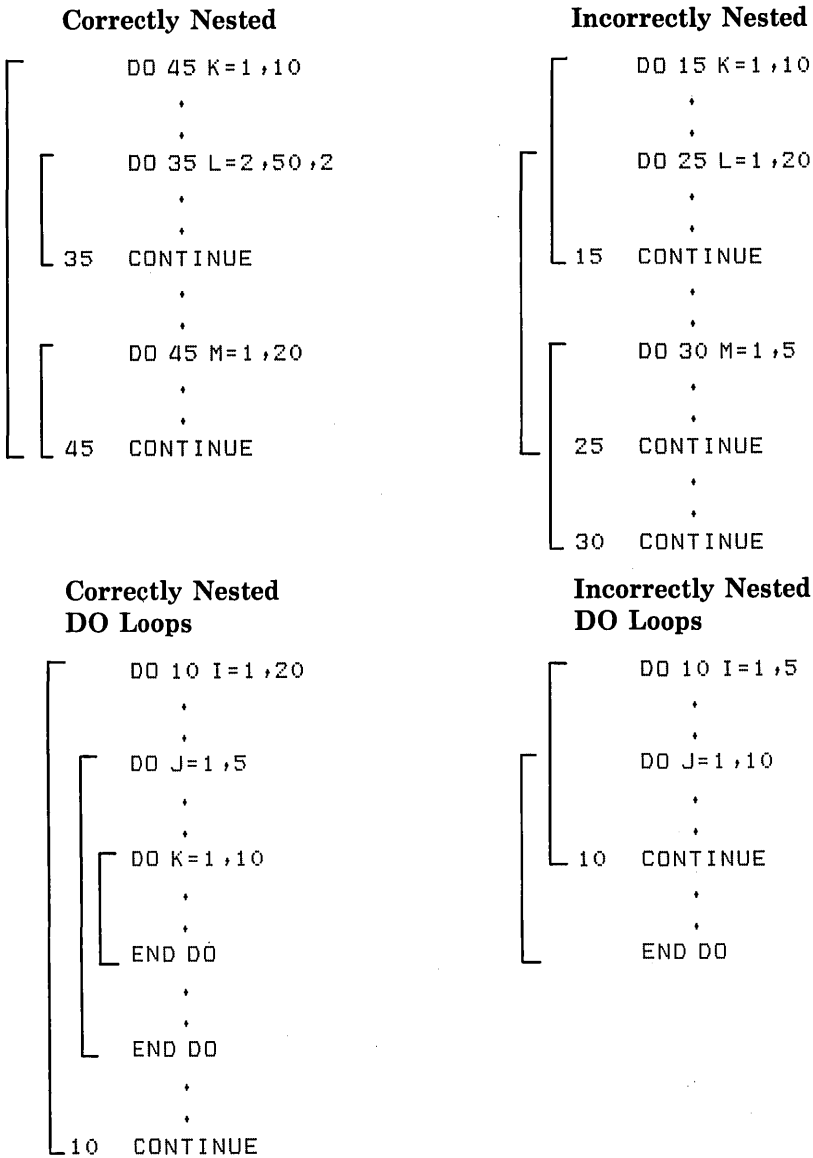


Figure 9-1: Nested DO Loops

9.3.1.3 Control Transfers in DO Loops

In a nested DO loop, you can transfer control from an inner loop to an outer loop. However, a transfer into a loop from outside that loop is not permitted.

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

9.3.1.4 Extended Range

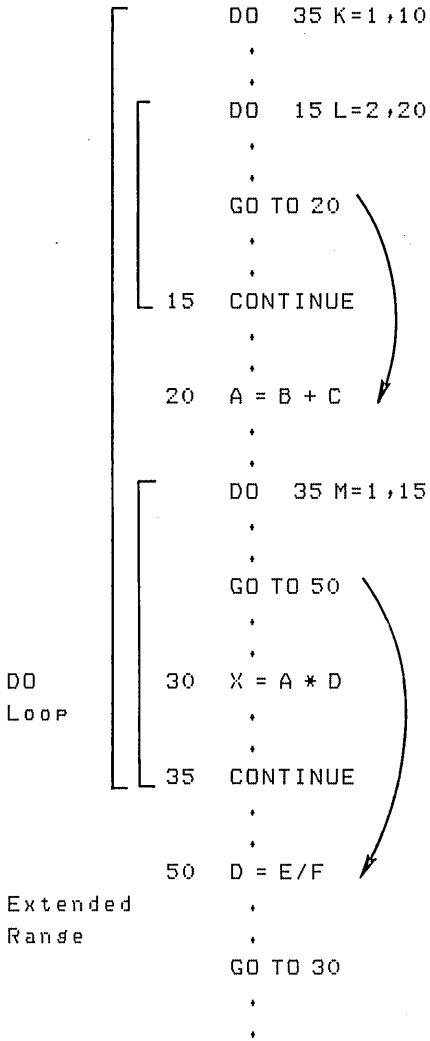
A DO loop has an extended range if it contains a control statement that transfers control out of the loop and if, after execution of one or more statements, another control statement returns control back into the loop. Thus, the range of the loop is extended to include all executable statements between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

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

1. 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.
2. The extended range of a DO statement must not change the control variable of the DO statement.

Figure 9-2 illustrates valid and invalid extended range control transfers.

**Valid
Control Transfers**



**Invalid
Control Transfers**

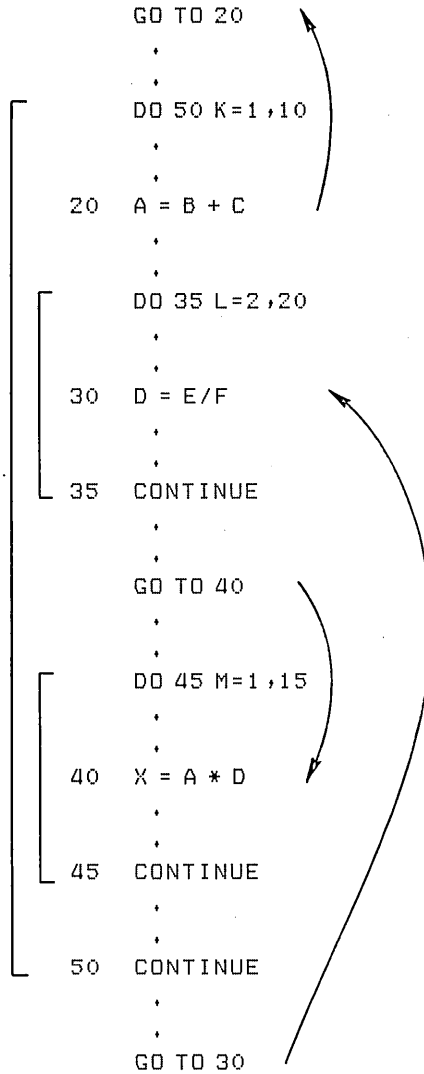


Figure 9-2: Control Transfers and Extended Range

9.3.2 DO WHILE Statement

The DO WHILE statement is similar to the DO statement discussed in the preceding section. The difference between them is as follows:

- The DO WHILE statement executes for as long as a logical expression contained in the statement continues to be true.
- The DO statement executes for a fixed number of iterations.

The DO WHILE statement has the form:

```
DO [s[,]] WHILE (e)
```

where:

s

is the label of an executable statement that must physically follow in the same program unit.

e

is a logical expression.

The DO WHILE statement tests the logical expression at the beginning of each execution of the loop, including the first. If the value of the expression is true, the statements in the body of the loop are executed; if the expression is false, control transfers to the statement following the loop.

If no label appears in a DO WHILE statement, the DO WHILE loop must be terminated with an END DO statement (see Section 9.4).

The following example demonstrates the use of the DO WHILE statement:

```
CHARACTER*132 LINE
I = 1
LINE(132:) = 'x'
DO WHILE (LINE(I:I) .EQ. ' ')
    I = I + 1
END DO
```

You can transfer control out of a DO WHILE loop but not into a loop from elsewhere in the program.

9.4 END DO Statement

The END DO statement terminates the range of a DO or DO WHILE statement. The END DO statement must be used to terminate a DO block, if the DO or DO WHILE statement defining the block does not contain a terminal-statement label. The END DO statement may also be used as a labeled terminal statement if the DO or DO WHILE statement does contain a terminal-statement label.

The END DO statement has the form:

```
END DO
```

Examples of the use of the END DO statement are:

```
DO WHILE (I .GT. J)          DO 10 WHILE (I .GT. J)
  ARRAY(I,J) = 1.0           ARRAY(I,J) = 1.0
  I = I - 1                  I = I - 1
END DO                        10 END DO
```

9.5 END Statement

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

The END statement has the form:

```
END
```

In a main program, if control reaches the END statement, program execution terminates. In a subprogram, a RETURN statement is implicitly executed.

If an initial line contains END in the statement field, and nothing else, it is treated as an END statement even if there are continuation lines that follow.

9.6 GO TO Statements

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

The three types of GO TO statement are:

- Unconditional GO TO
- Computed GO TO
- Assigned GO TO

9.6.1 Unconditional GO TO Statement

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

The unconditional GO TO statement has the form:

```
GO TO s
```

where:

s

is the label of an executable statement that is in the same program unit as the GO TO statement.

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

Examples of unconditional GO TO statements are:

```
GO TO 7734
```

```
GO TO 99999
```

9.6.2 Computed GO TO Statement

The computed GO TO statement transfers control to a statement based on the value of an expression within the statement.

The computed GO TO statement has the form:

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

where:

slist

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

e

is an arithmetic expression in the range 1 to n (where n is the number of statement labels in the transfer list).

The computed GO TO statement evaluates the expression e and, if necessary, converts the resulting value to integer data type. Control is transferred to the statement label in position e in the transfer list. For example, if the list contains (30,20,30,40), and the value of e is 2, control is transferred to statement 20.

If the value of e is less than 1 or greater than the number of labels in the transfer list, control is transferred to the first executable statement after the computed GO TO.

Examples of computed GO TO statements are:

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

```
GO TO (320,330,340,350,360), SITU(J,K) + 1
```


9.6.3 Assigned GO TO Statement

The assigned GO TO statement transfers control to a statement label that is represented by a variable. An ASSIGN statement must establish the relationship between the variable and a specific statement label. Thus, the transfer destination can be changed, depending on the most recently executed ASSIGN statement.

The assigned GO TO statement has the form:

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

where:

v

is an integer variable.

slist

is a list of one or more labels of executable statements separated by commas; slist does not affect statement execution and can be omitted.

The assigned GO TO statement transfers control to the statement whose label was most recently assigned to the variable v. The variable v must be of integer data type and must have a statement label value assigned to it by an ASSIGN statement (not an arithmetic assignment statement) before the GO TO statement is executed.

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

Examples of assigned GO TO statements are:

```
ASSIGN 200 TO IGO  
GO TO IGO
```

This is equivalent to GO TO 200.

```
ASSIGN 450 TO IBEG  
GO TO IBEG, (300,450,1000,25)
```

This is equivalent to GO TO 450.

9.7 IF Statements

IF statements conditionally transfer control or conditionally execute a statement or block of statements. The three types of IF statement are:

- Arithmetic IF
- Logical IF
- Block IF (IF THEN, ELSE IF THEN, ELSE, END IF)

For each type, the decision to transfer control or to execute the statement or block of statements is based on the evaluation of an expression within the IF statement.

9.7.1 Arithmetic IF Statement

The arithmetic IF statement conditionally transfers control to one of three statements, based on the current value of an arithmetic expression.

The arithmetic IF statement has the form:

```
IF (e) s1,s2,s3
```

where:

e

is an arithmetic expression.

s1,s2,s3

are labels of executable statements in the same program unit.

All three labels (s1,s2,s3) are required; however, they need not refer to three different statements.

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

If the value is:	Control passes to:
Less than 0	Label s1
Equal to 0	Label s2
Greater than 0	Label s3

Examples of arithmetic IF statements follow.

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

This statement transfers control to statement 50 if the real variable THETA is less than or equal to the real variable CHI. Control passes to statement 100 only if THETA is greater than CHI.

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

This statement transfers control to statement 40 if the value of the integer variable NUMBER is even; it transfers control to statement 20 if the value is odd.

9.7.2 Logical IF Statement

A logical IF statement conditionally executes a single FORTRAN statement, based on the current value of a logical expression within the logical IF statement.

The logical IF statement has the form:

```
IF (e) st
```

where:

e

is a logical expression.

st

is a complete FORTRAN statement. The statement can be any executable statement except a DO statement, an END DO statement, an END statement, a block IF statement, or another logical IF statement.

The logical IF statement first evaluates the logical expression **e**. If the value of the expression is true, the statement **st** is executed. If the value of the expression is false, control transfers to the next executable statement after the logical IF, and the statement **st** is not executed.

Examples of logical IF statements are:

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

```
IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K) * (-1.500)
```

```
IF (ENDRUN) CALL EXIT
```

9.7.3 Block IF Statements

Block IF statements conditionally execute blocks, or groups, of statements.

The four block IF statements are:

- IF THEN
- ELSE IF THEN
- ELSE
- END IF

These statements are used in block IF constructs. The block IF construct has the form:

```
IF (e) THEN  block

ELSE IF (e1) THEN
  block
  .
  .
  .
ELSE
  block

END IF
```

where:

e

is a logical expression.

block

is a sequence of zero or more complete FORTRAN statements. This sequence is called a statement block.

Figure 9-3 shows the flow of control for four examples of block IF constructs.

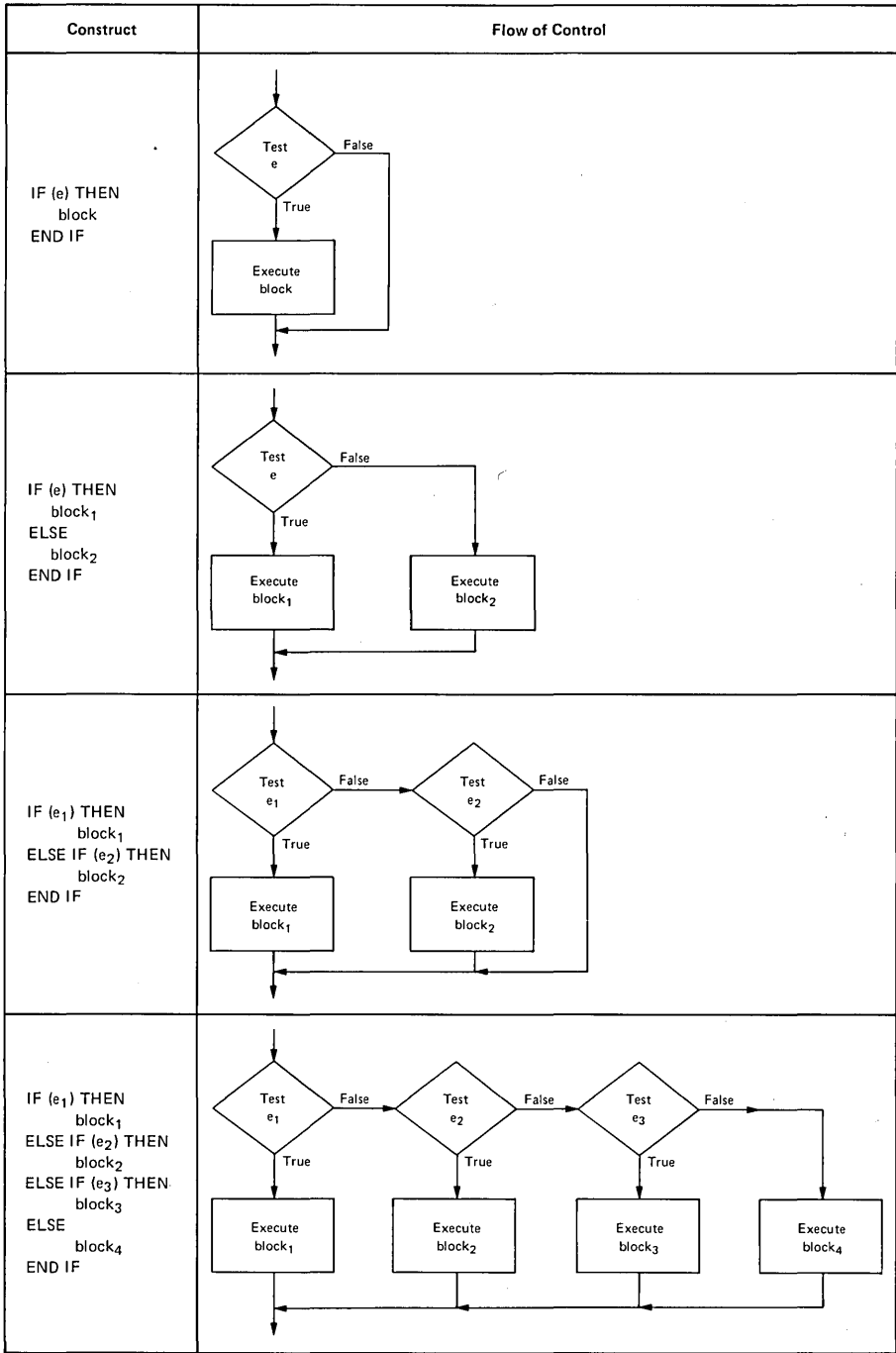
Each block IF statement, except the END IF statement, has an associated statement block. The statement block consists of all the statements following the block IF statement up to, but not including, the next block IF statement in the block IF construct. The statement block is conditionally executed based on the value(s) of the logical expression(s) in the preceding block IF statements.

The IF THEN statement begins a block IF construct. The block following it is executed if the value of the logical expression in the IF THEN statement is true.

The ELSE IF THEN statement is an optional statement that specifies a statement block to be executed if no preceding statement block in the block IF construct has been executed, and if the value of the logical expression in the ELSE IF THEN statement is true. A block IF construct can contain any number of ELSE IF THEN statements.

The ELSE statement specifies a statement block to be executed if no preceding statement block in the block IF construct has been executed. The ELSE statement is optional. However, if the ELSE statement is present, the ELSE statement block must be immediately followed by the END IF statement.

The END IF statement terminates the block IF construct.



ZK-617-82

Figure 9-3: Examples of Block IF Constructs

After the last statement in a statement block is executed, control passes to the next executable statement following the END IF statement. Consequently, no more than one statement block in a block IF construct is executed each time that the IF THEN statement is executed.

ELSE IF THEN and ELSE statements can have statement labels, but the labels cannot be referenced. The END IF statement can have a statement label to which control can be transferred, but only from within the immediately preceding block.

Section 9.7.3.1 describes restrictions on statements in a statement block. Section 9.7.3.2 describes examples of block IF constructs. Section 9.7.3.3 describes nested block IF constructs.

9.7.3.1 Statement Blocks

A statement block can contain any executable FORTRAN statement except an END statement. You can transfer control out of a statement block, but you must not transfer control into a block. Thus, you must not transfer control from one statement block to another.

DO loops cannot partially overlap statement blocks. When a statement block contains a DO statement, it must also contain the DO loop's terminal statement, and vice versa.

9.7.3.2 Block IF Examples

The following examples illustrate four variations of block IF constructs.

1. The simplest block IF construct consists of the IF THEN and END IF statements; this construct conditionally executes one statement block. For example:

Form	Example
IF (e) THEN block	IF (ABS(ADJU) .GE. 1.0E-6) THEN TOTERR = TOTERR + ABS(ADJU) QUEST = ADJU/FNDVAL
END IF	END IF

The statement block consists of all the statements between the IF THEN and the END IF statements.

The IF THEN statement first evaluates the logical expression e, or ABS(ADJU) .GE. 1.0E-6. If the value of e is true, the statement block is executed. If the value of e is false, control transfers to the next executable statement after the END IF statement, and the block is not executed.

2. The following example contains a block IF construct with an ELSE IF THEN statement:

Form	Example
IF (e1) THEN block1	IF (A .GT. B) THEN D = B F = A - B
ELSE IF (e2) THEN block2	ELSE IF (A .GT. B/2.) THEN D = B/2. F = A - B/2.
END IF	END IF

Block1 consists of all the statements between the IF THEN and the ELSE IF THEN statements; block2 consists of all the statements between the ELSE IF THEN and the END IF statements.

If A is greater than B, block1 is executed. If A is not greater than B but A is greater than B/2, block2 is executed. If A is not greater than B and A is not greater than B/2, neither block1 nor block2 is executed; control transfers directly to the next executable statement after the END IF statement.

3. The following example contains a block IF construct with an ELSE statement:

Form	Example
IF (e) THEN block1	IF (NAME .LT. 'N') THEN IFRONT = IFRONT + 1 FRLET(IFRONT) = NAME(1:2)
ELSE block2	ELSE IBACK = IBACK + 1
END IF	END IF

Block1 consists of all the statements between the IF THEN and ELSE statements; block2 consists of all the statements between the ELSE and the END IF statements.

If the value of the character variable NAME is less than 'N', block1 is executed. If the value of NAME is greater than or equal to 'N', block2 is executed.

4. The following example contains a block IF construct with several ELSE IF THEN statements and an ELSE statement:

Form	Example
IF (e1) THEN block1	IF (A .GT. B) THEN D = B F = A - B
ELSE IF (e2) THEN block2	ELSE IF (A .GT. C) THEN D = C F = A - C
ELSE IF (e3) THEN block3	ELSE IF (A .GT. Z) THEN D = Z F = A - Z
ELSE block4	ELSE D = 0.0 F = A
END IF	END IF

There are four statement blocks in this example. Each consists of all the statements between the block IF statements listed below.

Block	Delimiting Block IF Statements
block1	IF THEN and first ELSE IF THEN
block2	First ELSE IF THEN and second ELSE IF THEN
block3	Second ELSE IF THEN and ELSE
block4	ELSE and END IF

If A is greater than B, block1 is executed. If A is not greater than B but is greater than C, block2 is executed. If A is not greater than B or C but is greater than Z, block3 is executed. If A is not greater than B, C, or Z, block4 is executed.

9.7.3.3 Nested Block IF Constructs

A block IF construct can be included in a statement block of another block IF construct. But the nested block IF construct must be completely contained within a statement block; it must not overlap statement blocks.

The following example contains a nested block IF construct:

Form

```
IF (e1) THEN
  block1a
  ELSE
  block1b
  END IF
ELSE
  block2
  END IF
```

Example

```
IF (A .LT. 100) THEN
  INRAN = INRAN + 1
  IF (ABS(A-AVG) .LE. 5.) THEN
    INAVG = INAVG + 1
  ELSE
    OUTAVG = OUTAVG + 1
  END IF
ELSE
  OUTRAN = OUTRAN + 1
END IF
```

If A is less than 100, the code immediately after the IF is executed. This code contains a nested block IF construct. If the absolute value of A minus AVG is less than or equal to 5, block1a is executed. If the absolute value of A minus AVG is greater than 5, block1b is executed.

If A is greater than or equal to 100, block2 is executed, and the nested IF construct (block1) is not executed.

9.8 PAUSE Statement

The PAUSE statement displays a message on the terminal and temporarily suspends program execution in order to permit you to take some action.

The PAUSE statement has the form:

```
PAUSE [disp]
```

where:

disp

is a character constant or a string of decimal numbers (one to five digits).

The disp argument is optional. The effect of a PAUSE statement depends on how your program is being executed. If your program is running as a batch job or detached process, the contents of disp are written to the system output file, but the program is not suspended.

If the program is running in interactive mode, the contents of `disp` are displayed at your terminal, followed by the prompt sequence, indicating that the program is suspended and that you should enter a command. For example, if the following statement is executed in interactive mode:

```
PAUSE 'ERRONEOUS RESULT DETECTED'
```

you will see the following display at the terminal:

```
ERRONEOUS RESULT DETECTED  
$
```

If you do not specify a value for `disp`, the following message is displayed by the system:

```
FORTRAN PAUSE
```

You can respond by typing one of the following commands:

CONTINUE	—	Execution resumes at the next executable statement.
EXIT	—	Execution is terminated.
DEBUG	—	Execution resumes under control of the VAX Symbolic Debugger.

9.9 RETURN Statement

The `RETURN` statement transfers control from a subprogram to the program that called the subprogram. You can use `RETURN` statements only in subprogram units.

The `RETURN` statement has the form:

```
RETURN [i]
```

The optional argument, `i`, indicates an alternate return from the subprogram and can be specified only in subroutine subprograms; it cannot be specified in function subprograms. When specified, the value of `i` indicates that the `i`th alternate return in the actual argument list is to be taken (see the second example that follows in this section).

The value of `i` can be any integer constant or expression, for example, 2 or `I+J`. The system converts the type of the value to integer, if necessary.

When a `RETURN` statement is executed in a function subprogram, control is returned to the calling program at the statement that contains the function reference (see Chapter 10). When a `RETURN` statement is executed in a subroutine, control is returned either to the first executable statement following the `CALL` statement that initiated the subroutine, or to the statement label that was specified as the `i`th alternate return in the `CALL` argument list.

Examples of `RETURN` statements follow.

1. In the following example, control is returned to the calling program at the first executable statement following the CALL CONVRT statement.

```
SUBROUTINE CONVRT(N,ALPH,DATA,PRNT,K)
INTEGER ALPH(*), DATA(*), PRNT(*)
IF (N .GE. 10) THEN
    DATA(K+2) = N-(N/10)*N
    N = N/10
    DATA(K+1) = N
    PRNT(K+2) = ALPH(DATA(K+2)+1)
    PRNT(K+1) = ALPH(DATA(K+1)+1)
ELSE
    PRNT(K+2) = ALPH(N+1)
END IF
RETURN
END
```

2. The following example shows how alternate returns can be included in a subroutine.

```
SUBROUTINE CHECK(X,Y,*,*,C)
    .
    .
    .
50  IF (Z) 60,70,80
60  RETURN
70  RETURN 1
80  RETURN 2
END
```

If the value computed for Z is less than zero, a normal return is taken, and the calling program continues at the first executable statement following CALL CHECK. If Z equals zero, however, the first alternate return (RETURN 1) is taken; and if Z is greater than zero, the second alternate return (RETURN 2) is taken. Control is returned to the statement specified as the first or second alternate return argument in the CALL statement argument list. For example, if the CALL statement has the form:

```
CALL CHECK(A,B,*10,*20,C)
```

RETURN 1 transfers control to statement label 10, and RETURN 2 transfers control to statement label 20.

Note that if a subroutine includes an alternate return that specifies a value either less than 1 or greater than the number of alternate return arguments, control is returned to the next executable statement after the CALL statement. That is, the alternate returns are ignored. Therefore, you should ensure that the value of i is within the range of alternate return arguments.

9.10 STOP Statement

The STOP statement terminates program execution.

The STOP statement has the form:

```
STOP [disp]
```

where:

disp

is a character constant or a string of decimal numbers (one to five digits).

The disp argument is optional. If you specify it, the STOP statement displays the contents of disp at your terminal, terminates program execution, and returns control to the operating system. If you do not specify a value for disp, the following message is sent by the system:

```
FORTRAN STOP
```

Examples of STOP statements are:

```
STOP 98
```

```
STOP 'END OF RUN'
```


Chapter 10

Subroutines and Functions – Subprograms

Subprograms are program units that can be invoked from another program unit, usually to perform some commonly used computation on behalf of the other program unit. Subprograms are either supplied by the user or supplied as part of the VAX FORTRAN system.

User-supplied subprograms include the following:

- *Statement functions*—a computing procedure defined by a single statement that is similar in form to an assignment statement. A statement function is invoked by a function reference in a main program unit or a subprogram unit.
- *Function subprograms*—a program unit, also called a function, that contains a set of commonly used computations. A function subprogram's first statement is a FUNCTION statement, optionally preceded by an OPTIONS statement. A function subprogram is invoked by a function reference in a main program unit or a subprogram unit.
- *Subroutine subprograms*—a program unit, also called a subroutine, that contains a set of commonly used computations. A subroutine subprogram's first statement is a SUBROUTINE statement, optionally preceded by an OPTIONS statement. A subroutine subprogram receives control when it is invoked with a CALL statement and returns control with a RETURN statement.

Subprograms supplied with the FORTRAN system include the following:

- Intrinsic mathematical functions
- Intrinsic character functions
- Miscellaneous intrinsic functions

Normally, the program invoking the subprogram passes values, known as actual arguments, to the subprogram, which uses the actual arguments to compute the results and then returns the results to the calling program.

10.1 Subprogram Arguments

Subprogram arguments are either dummy arguments or actual arguments:

- Dummy arguments are specified when you write the subprogram.
- Actual arguments are specified when you invoke the subprogram.

When control is transferred to a subprogram, each dummy argument takes on the value of the corresponding actual argument. When control is returned to the calling program unit, the last value assigned to a dummy argument is assigned to the corresponding actual argument.

10.1.1 Actual Argument and Dummy Argument Association

Actual arguments must agree in order, number, and data type (or structure, for record arguments) with their corresponding dummy arguments. Actual arguments can be scalar references, array name references, aggregate references, alternate return specifiers, or subprogram names. The dummy arguments specified in subprogram definitions, representing corresponding actual arguments, appear as unsubscripted names.

Although dummy arguments are not actual variables, arrays, records, or subprograms, each dummy argument can be declared as though it were a variable, array, record, or subprogram.

- A dummy argument declared as an array can be associated only with an actual argument that is an array or array element of the same data type. The actual argument must not be placed in parentheses. If a dummy argument is an array, it must be no larger than the array that is the actual argument. You can use adjustable arrays (see Section 10.1.1.1) to process arrays of different sizes in a single subprogram.
- A dummy argument declared as a record can be associated only with an actual argument that is an aggregate reference for an entity with a matching structure.
- A dummy argument referenced as a subprogram must be associated with an actual argument that has been declared `EXTERNAL` or `INTRINSIC` in the calling routine.

The length of a dummy argument with a data type of character must not be greater than the length of its associated actual argument. Note that if the character dummy argument's length is specified as `*(*)`, the length used is exactly the length of the associated actual argument. (This is known as a passed-length character argument. See Section 10.1.1.3.)

The following topics are discussed in Sections 10.1.1.1 through 10.1.1.5.

- Adjustable array arguments
- Assumed-size array arguments
- Passed-length character arguments
- Character and Hollerith constants as actual arguments
- Alternate return arguments

10.1.1.1 Adjustable Arrays

Adjustable arrays are dummy arguments in subprograms. The dimensions of an adjustable array are determined in the reference to the subprogram. The array declarator (see Section 6.2.3.1) for an adjustable array can contain integer variables that are either dummy arguments or variables in a common block.

When the subprogram is entered, each dummy argument used in the array declarator must be associated with an actual argument, and each variable in a common block used in an array declarator must have a defined value. The dimension declarator is evaluated using the values of the actual arguments, variables in common blocks, and constants specified in the array declarator.

The size of the adjustable array must be less than or equal to the size of the array that is its corresponding actual argument.

The function in the following example computes the sum of the elements of a two-dimensional array. Note the use of the dummy arguments M and N to control the iteration.

```
FUNCTION SUM(A,M,N)
  DIMENSION A(M,N)
  SUM = 0.0
  DO 10 J=1,N
  DO 10 I=1,M
10  SUM = SUM + A(I,J)
  RETURN
END
```

The following statements are sample calls on SUM:

```
DIMENSION A1(10,35), A2(3,56)
SUM1 = SUM(A1,10,35)
SUM2 = SUM(A2,3,56)
SUM3 = SUM(A1,10,10)
```

The upper- and lower-dimension bound values are determined once each time a subprogram is entered. These values do not change during the execution of that subprogram even if the values of variables contained in the array declaration are changed. For example:

```
DIMENSION ARRAY(9,5)
L = 9
M = 5
CALL SUB(ARRAY,L,M)
END

SUBROUTINE SUB(X,I,J)
DIMENSION X(-I/2:I/2,J)
X(I/2,J) = 999
J = 1
I = 2
END
```

In this example, the adjustable array X is declared as X(-4:4,5) on entry to subroutine SUB. The assignments to I and J do not affect that declaration.

Once a variable is used in an array declarator for an adjustable array, it must not appear in a type declaration that changes the variable's data type. Thus, the following program segment is invalid:

```
SUBROUTINE SUB1(A,X)
DIMENSION A(X)
INTEGER X
```

An adjustable array is undefined if a dummy argument array is not currently associated with an actual argument array. It is also undefined if any of the variables in the adjustable array declarator are either not currently associated with an actual argument or not in a common block. Note that argument association is not retained between one reference to a subprogram and the next reference to that subprogram.

This is illustrated by the following example of a subroutine subprogram and the sample statements of a calling program unit:

1. Subroutine subprogram:

```
SUBROUTINE S(A,I,X)
DIMENSION A(I)
A(I) = X
RETURN
ENTRY S1(I,A,K,L)
A(I) = A(I) + 1.0
RETURN
END
```

2. Statements in the program unit that calls the subroutine subprogram:

```
DIMENSION B(10)
.
.
.
CALL S(B,2,3.0)
.
.
CALL S1(5,B,3,2)
```

In the preceding example, the calling program unit defines B as a real array with 10 elements. The first call to subroutine S sets array element B(2) equal to the value 3.0. The second call to subroutine S (at entry point S1) increments array element B(5) by the value 1.0. RECORD statements not contained within structure declaration blocks can also declare adjustable arrays.

10.1.1.2 Assumed-Size Arrays

An assumed-size array is a dummy array for which the upper bound of the last dimension is specified as an asterisk (*). For example:

```
SUBROUTINE SUB(A,N)
DIMENSION A(1:N,1:*)
.
.
.
```

The size of an assumed-size array and the number of elements that can be referenced are determined as follows:

- If the actual argument corresponding to the dummy array is a name of a noncharacter array, the size of the dummy array is the size of the actual-argument array.
- If the actual argument corresponding to the dummy argument is a name of a non-character array element, with a subscript value of s in an array of size a , the size of the dummy array is $a+1-s$.
- If the actual argument is a name of a character array, character array element, or character array element substring and begins at character storage unit b of an array with n character storage units, the size of the dummy array is $\text{INT}(n+1-b)/y$, where y is the length of an element of the dummy array.

Because the actual size of an assumed-size array is not known, an assumed-size array name cannot be used as any of the following:

- An array name in the list of an I/O statement
- A unit identifier for an internal file in an I/O statement
- A run-time format specifier in an I/O statement

RECORD statements not contained within structure declaration blocks can also declare adjustable arrays.

10.1.1.3 Passed-Length Character Arguments

A passed-length character argument is a dummy argument that assumes the length attribute of the corresponding actual argument. An asterisk is used to specify the length of the dummy character argument.

When control transfers to the subprogram, each dummy argument assumes the length of its corresponding actual argument.

A character array dummy argument can also have a passed length. The length of each element in the dummy argument is the length of the elements in the actual argument. The passed length and the array declarator together determine the size of the passed-length character array. A passed-length character array can also be an adjustable or assumed-size array.

The following example of a function subprogram uses a passed-length character argument. The function finds the position of the character with the highest ASCII code value; it uses the length of the passed-length character argument to control the iteration. (Note that the processor-defined function LEN is used to determine the length of the argument. See Section 10.3.4.1 for a description of the LEN function.)

```
      INTEGER FUNCTION ICMAX(CVAR)
      CHARACTER*(*) CVAR
      ICMAX = 1
      DO 10 I=2,LEN(CVAR)
10    IF (CVAR(I:I) .GT. CVAR(ICMAX:ICMAX)) ICMAX=I
      RETURN
      END
```

The length of the dummy argument is determined each time control transfers to the function. The length of the actual argument can be the length of a character variable, array element, substring, or expression. Each of the following function references specifies a different length for the dummy argument:

```
CHARACTER VAR*10, CARRAY(3,5)*20
.
.
.
I1 = ICMAX(VAR)
I2 = ICMAX(CARRAY(2,2))
I3 = ICMAX(VAR(3:8))
I4 = ICMAX(CARRAY(1,3)(5:15))
I5 = ICMAX(VAR(3:4)//CARRAY(3,5))
```

10.1.1.4 Character and Hollerith Constants as Actual Arguments

Actual arguments and their corresponding dummy arguments must agree in data type. If the actual argument is a Hollerith constant (for example, 4HABCD), the dummy argument must be of numeric data type. In VAX FORTRAN, if an actual argument is a character constant (for example, 'ABCD'), the corresponding dummy argument can have either a numeric or a character data type. If the dummy argument has a numeric data type, the character constant 'ABCD' is, in effect, converted to a Hollerith constant by the FORTRAN compiler and the linker.

An exception to this occurs when the function or subroutine name is itself a dummy argument. It is not possible to determine at compile time or link time whether a character constant or Hollerith constant is required. In this case, a character constant actual argument can correspond only to a character dummy argument. For example:

```
SUBROUTINE S(CHARSUB,HOLLSUB,A,B)
EXTERNAL CHARSUB,HOLLSUB
.
.
.
CALL CHARSUB(A,'STRING')
CALL HOLLSUB(B,6HSTRING)
```

In this example, the subroutine names CHARSUB and HOLLSUB are themselves dummy arguments of the subroutine S. Therefore, the actual argument 'STRING' in the call to CHARSUB must correspond to a character dummy argument, whereas the actual argument 6HSTRING in the call to HOLLSUB must correspond to a Hollerith dummy argument.

10.1.1.5 Alternate Return Arguments

To specify an alternate return argument in a dummy argument list, place asterisks in the list. For example:

```
SUBROUTINE MINN(A,B,*,*,C)
```

The actual argument list passed in the CALL must include alternate return arguments in the corresponding positions. These arguments have the form:

*label
or
&label

You can use either an asterisk or an ampersand to indicate an alternate return argument in an actual argument list. The value you specify for label must be the label of an executable statement in the program that issued the CALL.

10.1.2 Built-In Functions

Built-in functions perform utility operations that are useful in communicating with subprograms written in languages other than FORTRAN. The two kinds of built-in functions are:

- Argument list built-in functions
- %LOC built-in function

10.1.2.1 Argument List Built-In Functions

To call subprograms (such as VAX/VMS system services) written in languages other than FORTRAN, you may need to pass the actual arguments in a form different from that used by FORTRAN. To change the form of the argument, you can use the built-in functions %VAL, %REF, and %DESCR in the argument list of a CALL statement or function reference. These built-in functions specify the way the argument should be passed to the subprogram. You can use them only in the actual argument list of a CALL statement or function reference. You cannot use them in any other context.

The argument list built-in functions are:

Function	Effect
%VAL(a)	Pass the argument as a 32-bit immediate value (if the actual argument is shorter than 32 bits, it is sign-extended to a 32-bit value)
%REF(a)	Pass the argument by reference
%DESCR(a)	Pass the argument by descriptor

In these functions, a is an actual argument.

See the *VAX FORTRAN User's Guide* for more information on argument-passing mechanisms.

Table 10-1 lists the FORTRAN argument-passing defaults and the allowed uses of %VAL, %REF, and %DESCR.

Table 10-1: Argument List Built-In Functions and Defaults

Actual Argument Data Type	Default	Functions Allowed		
		%VAL	%REF	DESCR
Expressions				
Logical	REF	Yes ¹	Yes	Yes
Integer	REF	Yes ¹	Yes	Yes
REAL*4	REF	Yes	Yes	Yes
REAL*8	REF	No	Yes	Yes
REAL*16	REF	No	Yes	Yes
Complex	REF	No	Yes	Yes
Character	DESCR	No	Yes	Yes
Holerith	REF	No	No	No
Aggregate	REF	No	Yes	No
Array Name				
Numeric	REF	No	Yes	Yes
Character	DESCR	No	Yes	Yes
Aggregate	REF	No	Yes	No
Procedure Name				
Numeric	REF	No	Yes	Yes
Character	DESCR	No	Yes	Yes

¹ If a logical or integer value occupies less than 32 bits of storage, it is converted to a 32-bit value by sign extension. Use the ZEXT function if zero extension is desired.

10.1.2.2 %LOC Built-In Function

The %LOC built-in function computes the internal address of a storage element. It has the form:

%LOC(arg)

where:

arg

is a scalar memory reference, array name reference, aggregate reference, or external procedure name.

The %LOC built-in function produces an INTEGER*4 value that represents the location of its argument. The INTEGER*4 value can be used as an element in an arithmetic expression.

See the *VAX FORTRAN User's Guide* for more information on the %LOC built-in function.

10.2 User-Written Subprograms

A user-written subprogram is a FORTRAN statement or group of FORTRAN statements that performs a computing procedure. The computing procedure can be either a series of arithmetic operations or a series of FORTRAN statements. You can use a single subprogram to perform a computing procedure in several places in your program, and thus avoid duplicating the series of operations or statements in each place.

There are three types of subprograms. Table 10-2 lists each type of subprogram, the statements needed to define the subprogram, and the method of transferring control to it.

Table 10-2: Types of User-Written Subprogram

Subprogram Type	Defining Statements	Control Transfer Method
Statement function	Statement function definition	Function reference
Function	FUNCTION ENTRY	Function reference
Subroutine	SUBROUTINE ENTRY	CALL statement

A function reference is used in an expression and consists of the function name and the function arguments. A function reference returns a value that is used in evaluating the expression in which the function appears.

Function and subroutine subprograms can change the values of their arguments, and the calling program can use the changed values.

A subprogram can refer to other subprograms; but it cannot, either directly or indirectly, refer to itself.

10.2.1 Statement Functions

A statement function is a computing procedure defined in the same program unit in which it is referenced. It is defined by a single statement that is similar in form to an assignment statement. The computation is performed each time you refer to the statement function. The resulting value is then made available to the expression that contains the statement function reference.

The statement function definition statement has the form:

$$\text{fun}([\text{p},\text{p}]\dots) = \text{e}$$

where:

fun

is the symbolic name of the statement function.

p

is a dummy argument.

e

is an expression.

The expression (e) is an arithmetic, logical, or character expression that defines the computation to be performed.

A statement function reference has the form:

$$f([\text{p},\text{p}]\dots)$$

where:

f

is the symbolic name of the function.

p

is an actual argument.

Rules governing the use of statement function definitions and references are as follows:

- When a statement function reference appears in an expression, the values of the actual arguments are associated with the dummy arguments in the statement function definition. The expression in the definition is then evaluated. The resulting value is used to complete the evaluation of the expression containing the function reference.
- The data type of a statement function is determined either implicitly by the initial letter of the function name, or explicitly in a type declaration statement. The data type can be any of the data types, including the character data type.
- Dummy arguments in a statement function indicate only the number, order, and data type of the actual arguments. You can use the names of the dummy arguments to represent other entities elsewhere in the program unit. Note that, except for data type, declarative information associated with an entity is not associated with the dummy arguments in the statement function. That is, declaring an entity to be an array or to be in a common block does not affect a dummy argument with the same name.
- Actual arguments must agree in number, order, and data type with their corresponding dummy arguments.

- You cannot use the name of the statement function to represent any other entity within the same program unit.
- The expression in a statement function definition can contain function references. If a reference to another statement function appears in the expression, you must have previously defined that function in the same program unit.
- Any reference to a statement function must appear in the same program unit as the definition of that function.
- A statement function reference must appear as, or be part of, an expression. You cannot use the reference as the left side of an assignment statement.

Examples

- Examples of statement function definitions:

```
VOLUME(RADIUS) = 4.189*RADIUS**3
```

```
SINH(X) = (EXP(X)-EXP(-X))*0.5
```

```
CHARACTER*10 CSF,A,B
CSF(A,B) = A(6:10)//B(1:5)
```

The following definition is invalid. A constant cannot be used as a dummy argument.

```
AVG(A,B,C,3.) = (A+B+C)/3.
```

- Examples of statement function references:

Given the definition:

```
AVG(A,B,C) = (A+B+C)/3.
```

The references are:

```

      *
      *
      *
GRADE = AVG(TEST1,TEST2,XLAB)
IF (AVG(P,D,Q) .LT. AVG(X,Y,Z)) GO TO 300
```

The following reference is invalid. The data type of the third argument does not agree with the dummy argument.

```
FINAL = AVG(TEST3,TEST4,LAB2)
```

10.2.2 Function Subprograms

A function subprogram is a program unit consisting of a **FUNCTION** statement followed by a series of statements that define a computing procedure. You use a function reference to transfer control to a function subprogram, and a **RETURN** or **END** statement to return control to the calling program unit.

A function subprogram returns a single value to the calling program unit by assigning that value to the function's name. The function's name determines the data type of the value returned.

10.2.2.1 Logical and Numeric Functions

The FUNCTION statement has the form:

```
[typ] FUNCTION nam[*m]([(p[,p]...)])
```

where:

typ

is one of the logical or numeric data type specifiers. See Section 8.4.1 for a list of these specifiers.

nam

is the symbolic name of the function.

m

is an unsigned, nonzero integer constant specifying the length of the data type; it must be one of the valid length specifiers for the data type given by typ.

p

is a dummy argument.

10.2.2.2 Character Functions

The CHARACTER FUNCTION statement has the form:

```
CHARACTER[*n] FUNCTION nam[*n]([(p[,p]...)])
```

where:

n

is an unsigned, nonzero integer constant, or parenthetical asterisk (*) indicating a passed-length function name. If you specify CHARACTER*(*), the function assumes the length declared for it in the program unit that invokes it. A passed-length character function can have different lengths when it is invoked by different program units. If n is an integer constant, the value of n must agree with the length of the function specified in the program unit that invokes the function. If you do not specify n, a length of one is assumed. If the length has already been specified following the keyword CHARACTER, the optional length specification following nam is not permitted.

nam

is the symbolic name of the function.

p

is a dummy argument.

10.2.2.3 Function Reference

A function reference that transfers control to a function subprogram has the form:

nam([p[,p]...])

where:

nam

is the symbolic name of the function.

p

is an actual argument.

When control transfers to a function subprogram, the values of the actual arguments (if any) in the function reference are associated with the dummy arguments (if any) in the FUNCTION statement. The statements in the subprogram are then executed and the resulting value is assigned to the name of the function. Finally, the function returns control to the calling program unit. The value assigned to the function's name is now available to the expression containing the function reference and is used to complete the evaluation of that expression.

The data type of a function name can be specified explicitly in the FUNCTION statement or in a type declaration statement, or it can be specified implicitly. The function name defined in the function subprogram must have the same data type as the function name in the calling program unit.

The FUNCTION statement must be the first statement of a function subprogram, unless an OPTIONS statement is used. A function subprogram cannot contain a SUBROUTINE statement, a BLOCK DATA statement, a PROGRAM statement, or another FUNCTION statement. ENTRY statements can be included to provide multiple entry points to the subprogram (see Section 10.2.4).

Examples of function subprograms follow.

1. In the following example

```
FUNCTION ROOT(A)
  X = 1.0
2  EX = EXP(X)
  EMINX = 1./EX
  ROOT = ((EX+EMINX)*.5+COS(X)-A)/((EX-EMINX)*.5-SIN(X))
  IF (ABS(X-ROOT) .LT. 1E-6) RETURN
  X = ROOT
  GO TO 2
END
```

the function uses the Newton-Raphson iteration method to obtain the root of the function:

$$F(X) = \cosh(X) + \cos(X) - A = 0$$

The value of A is passed as an argument. The iteration formula for this root is:

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

This is calculated repeatedly until the difference between X_i and X_{i+1} is less than $1.0E-6$. The function uses the FORTRAN intrinsic functions EXP, SIN, COS, and ABS (see Section 10.3).

2. The following example is a passed-length character function. It returns the value of its argument, repeated to fill the length of the function.

```
CHARACTER*(*) FUNCTION REPEAT(CARG)
CHARACTER*1 CARG
DO 10 I=1,LEN(REPEAT)
10 REPEAT(I:I) = CARG
RETURN
END
```

3. Within any given program unit all references to a passed-length character function must have the same length. In the following example, the REPEAT function has a length of 1000:

```
CHARACTER*1000 REPEAT, MANYAS, MANYZS
MANYAS = REPEAT('A')
MANYZS = REPEAT('Z')
```

However, another program unit within the executable program can specify a different length. In the following example, the REPEAT function has a length of 2:

```
CHARACTER HOLD*6, REPEAT*2
HOLD = REPEAT('A')//REPEAT('B')//REPEAT('C')
```

10.2.3 Subroutine Subprograms — SUBROUTINE Statement

A subroutine subprogram is a program unit consisting of a SUBROUTINE statement followed by a series of statements that define a computing procedure. You use a CALL statement to transfer control to a subroutine subprogram, and a RETURN or END statement to return control to the calling program unit.

The SUBROUTINE statement has the form:

```
SUBROUTINE sub [(p[,p]...)]
```

where:

sub

is the symbolic name of the subroutine.

p

is a dummy argument. You can specify a dummy argument as an alternate return argument by placing an asterisk in the argument list.

Section 9.1 describes the CALL statement.

When control transfers to the subroutine, the values of the actual arguments (if any) in the CALL statement are associated with the corresponding dummy arguments (if any) in the SUBROUTINE statement. The statements in the subprogram are then executed.

The SUBROUTINE statement must be the first statement of a subroutine, unless an OPTIONS statement is used.

A subroutine subprogram cannot contain a FUNCTION statement, a BLOCK DATA statement, a PROGRAM statement, or another SUBROUTINE statement. ENTRY statements are allowed to specify multiple entry points in the subroutine (see Section 10.2.4).

Examples

The following example contains a subroutine that computes the volume of a regular polyhedron, given the number of faces and the length of one edge. It uses the computed GO TO statement to determine whether the polyhedron is a tetrahedron, cube, octahedron, dodecahedron, or icosahedron. The GO TO statement also transfers control to the proper procedure for calculating the volume. If the number of faces is not 4, 6, 8, 12, or 20, the subroutine sends an error message to the user's terminal.

Main Program

```
COMMON NFACES , EDGE , VOLUME
ACCEPT * , NFACES , EDGE
CALL PLYVOL
TYPE * , 'VOLUME=' , VOLUME
STOP
END
```

Subroutine

```
      SUBROUTINE PLYVOL
      COMMON NFACES , EDGE , VOLUME
      CUBED = EDGE**3
      GO TO (6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,6,5) , NFACES
      GO TO 6
1     VOLUME = CUBED * 0.11785
      RETURN
2     VOLUME = CUBED
      RETURN
3     VOLUME = CUBED * 0.47140
      RETURN
4     VOLUME = CUBED * 7.66312
      RETURN
5     VOLUME = CUBED * 2.18170
      RETURN
6     TYPE 100 , NFACES
100  FORMAT ( ' NO REGULAR POLYHEDRON HAS ',I3,' FACES. '//)
      VOLUME = 0.0
      RETURN
      END
```

The following example illustrates the use of alternate return specifiers to determine where control is to be transferred on completion of the subroutine. The SUBROUTINE statement argument list contains two dummy alternate return arguments corresponding to the actual arguments *10 and *20 in the CALL statement argument list. The decision about which RETURN statement to execute depends on the value of Z, as computed in the subroutine. Thus, if Z is less than zero, the normal return is taken; if Z is equal to zero, the return is to statement label 10 in the main program; if Z is greater than zero, the return is to statement label 20 in the main program.

Main Program

```
CALL CHECK(A,B,*10,*20,C)
TYPE *, 'VALUE LESS THAN ZERO'
GO TO 30
10 TYPE *, 'VALUE EQUALS ZERO'
GO TO 30
20 TYPE *, 'VALUE MORE THAN ZERO'
30 CONTINUE
.
.
.
```

Subroutine

```
SUBROUTINE CHECK(X,Y,*,*,Q)
.
.
.
50 IF (Z) 60,70,80
60 RETURN
70 RETURN 1
80 RETURN 2
END
```

10.2.4 ENTRY Statement

The ENTRY statement provides multiple entry points within a subprogram. It is not executable and can appear within a function or subroutine program after the FUNCTION or SUBROUTINE statement. Execution of a subprogram referred to by an entry name begins with the first executable statement after the ENTRY statement.

The ENTRY statement has the form:

```
ENTRY nam([(p[,p]...)])
```

where:

nam

is the symbolic name of an entry point.

p

is a dummy argument.

Considerations/Restrictions

- Use the CALL statement to refer to entry names within subroutine subprograms. Use function references to refer to entry names within function subprograms.
- An entry name within a function subprogram can appear in a type declaration statement.

- You can specify an entry name in an `EXTERNAL` statement and use it as an actual argument; you cannot use it as a dummy argument.
- You cannot use entry names in executable statements that physically precede the appearance of the entry name in an `ENTRY` statement.
- You can include alternate return arguments in `ENTRY` statements by placing asterisks in the dummy argument list. `ENTRY` statements that specify alternate return arguments can be used only in subroutine subprograms.
- You can use dummy arguments in `ENTRY` statements that differ in order, number, type, and name from the dummy arguments you use in the `FUNCTION`, `SUBROUTINE`, and other `ENTRY` statements in the same subprogram. However, each reference to a function, subroutine, or entry must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding `FUNCTION`, `SUBROUTINE`, or `ENTRY` statement.
- A dummy argument can be referred to only in executable statements that follow the first `SUBROUTINE`, `FUNCTION`, or `ENTRY` statement in which the dummy argument is specified. A dummy argument is undefined if it is not currently associated with an actual argument. An argument association is not retained from one reference of a subprogram to the next.
- You cannot use an `ENTRY` statement within a block `IF` construct or a `DO` loop.

10.2.4.1 ENTRY in Function Subprograms

All entry names within a function subprogram are associated with the name of the function subprogram. Therefore, defining any entry name or the name of the function subprogram defines all the associated names of the same data type; all associated names that are of different data types become undefined. The function and entry names do not need to be of the same data type, but they all must be consistent within one of the following groups of data types:

Group 1: `BYTE`, `INTEGER*2`, `INTEGER*4`, `LOGICAL*2`, `LOGICAL*4`, `REAL*4`, `REAL*8`, `COMPLEX*8`

Group 2: `COMPLEX*16`, `REAL*16`

Group 3: `CHARACTER`

When either a `RETURN` statement or the implied return at the end of a subprogram is executed, the symbolic name used to refer to the function subprogram must be defined.

If the function is of character data type, all entry names must also be of character data type and must have the same length specification as that of the function. Note that the length specified must also agree with the length specified in the program unit referring to the entry name. If an asterisk enclosed in parentheses is used to specify the length of the entry name, the entry name has a passed length (see Section 10.1.1.3).

Figure 10-1 illustrates a function subprogram that computes the hyperbolic functions sinh, cosh, and tanh.

```
REAL FUNCTION TANH(X)
C   Statement function to compute twice sinh
    TSINH(Y) = EXP(Y) - EXP(-Y)
C   Statement function to compute twice cosh
    TCOSH(Y) = EXP(Y) + EXP(-Y)
C   Compute tanh
    TANH = TSINH(X)/TCOSH(X)
    RETURN
C   Compute sinh
    ENTRY SINH(X)
    SINH = TSINH(X)/2.0
    RETURN
C   Compute cosh
    ENTRY COSH(X)
    COSH = TCOSH(X)/2.0
    RETURN
END
```

Figure 10-1: Multiple Functions in a Function Subprogram

10.2.4.2 ENTRY in Subroutine Subprograms

To refer to an entry point name in a subroutine, issue a CALL statement that includes the entry point name defined in the ENTRY statement. For example:

Main Program

```
CALL SUBA(A,B,C)
```

```
·
·
·
```

Subroutine

```
SUBROUTINE SUB(X,Y,Z)
```

```
·
·
·
```

```
ENTRY SUBA(Q,R,S)
```

In this example, the CALL is to an entry point (SUBA) within the subroutine (SUB). Execution begins with the first statement following ENTRY SUBA (Q,R,S), using the actual arguments (A,B,C) passed in the CALL statement. Note that alternate returns can be specified in ENTRY statements. For example:

```
SUBROUTINE SUB(K,*,*)  
  .  
  .  
  .  
  ENTRY SUBC(J,K,*,*,X)  
  .  
  .  
  .  
  RETURN 1  
  RETURN 2  
  END
```

If you issue a CALL to entry point SUBC, you must include actual alternate return arguments. For example:

```
CALL SUBC(M,N,*100,*200,P)
```

In this case, RETURN 1 transfers control to statement label 100 and RETURN 2 transfers control to statement label 200 in the calling program.

10.3 FORTRAN Intrinsic Functions

Intrinsic functions, supplied in the VAX/VMS FORTRAN library, perform commonly used mathematical computations.

Function references to FORTRAN intrinsic functions are written in the same way that function references to user-defined functions are written. For example:

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

As a result of this reference, the absolute value of X-1 is calculated and multiplied by the constant 3.14159; the result is assigned to the variable R.

Appendix D lists the intrinsic functions, their data types, and the data types of their actual arguments. For descriptions of the intrinsic function algorithms, refer to the *VAX/VMS Run-Time Library Reference Manual*.

The two methods of referencing intrinsic functions are described in the sections that follow.

10.3.1 Intrinsic Function References

FORTRAN library function names are called intrinsic function names. Normally, a name in the table of intrinsic function names (Table D-1) refers to the FORTRAN library function with that name. However, the name can refer to a user-defined function when the name appears in an EXTERNAL statement (see Section 8.7).

Except when they are used in an `EXTERNAL` statement, intrinsic function names are local to the program unit that refers to them. Thus, they can be used for other purposes in other program units. In addition, the data type of an intrinsic function does not change if you use an `IMPLICIT` statement to change the implied data type rules.

Note that you cannot have an intrinsic function and a user-defined function with the same name in the same program unit.

10.3.2 Generic Function References

Many of the intrinsic functions supplied with VAX FORTRAN are generic functions, which means that you refer to them by a common name and the selection of the actual library routine to be used is based on the data type of the argument in the function reference. For example, there are five intrinsic functions that calculate cosines. All of them can be referred to by the generic name `COS`. Their names are `COS`, `DCOS`, `QCOS`, `CCOS`, and `CDCOS`. These functions differ in that they return `REAL*4`, `REAL*8`, `REAL*16`, `COMPLEX*8`, and `COMPLEX*16` values, respectively. To invoke the cosine function, you can refer to it generically as `COS`, and the compiler selects the appropriate routine, based on the arguments that you specify. For example, if the argument is `REAL*4`, `COS` is selected; if it is `REAL*8`, `DCOS` is selected; and if `COMPLEX*8`, `CCOS` is selected.

Note, however, that you can explicitly refer to a particular routine if you wish. Thus, to invoke the double-precision cosine function, you could specify `DCOS` rather than use the generic name.

The compiler lists the internal names of the intrinsic functions it has selected in the “FUNCTIONS AND SUBROUTINES REFERENCED” section of the listing.

Generic function selection occurs independently for each function reference. Thus, you can use a generic function reference repeatedly, in the same program unit, to access different intrinsic functions.

Table 10-3 lists the generic function names. If you use the names in Table 10-3 in any of the following ways, you cannot use them for generic function selection:

- As the name of a statement function
- As a dummy argument name, a common block name, or a variable or array name

Using a generic name in an `INTRINSIC` statement (see Section 8.9) does not affect function references. When you use a generic function name in an actual argument list as the name of a function to be passed, generic function selection does not occur because there is no argument list on which to base a selection. The name is treated according to the rules for nongeneric FORTRAN functions described in Section 10.3.

Generic function names are local to the program unit that refers to them. Thus, they can be used for other purposes in other program units.

Table 10-3: Generic Function Name Summary

Generic Name	Data Type of Argument	Data Type of Result
ABS	Integer	Integer
	Real	Real
	COMPLEX*8	REAL*4
	COMPLEX*16	REAL*8
AINT, ANINT	Real	Real
NINT	Real	Integer
INT	Integer	Integer
	Real	Integer
	Complex	Integer
REAL	Integer	REAL*4
	Real	REAL*4
	Complex	REAL*4
DBLE	Integer	REAL*8
	Real	REAL*8
	Complex	REAL*8
QEXT	Integer	REAL*16
	Real	REAL*16
	Complex	REAL*16
CMPLX	Integer	COMPLEX*8
	Real	COMPLEX*8
	Complex	COMPLEX*8
DCMPLX	Integer	COMPLEX*16
	Real	COMPLEX*16
	Complex	COMPLEX*16
MOD, MAX, MIN, SIGN, DIM	Integer	Integer
	Real	Real
EXP, LOG, SIN, COS, SQRT	Real	Real
	Complex	Complex
LOG10, SIND, COSD, TAN, TAND, ATAN, ATAND, ATAN2, ATAN2D, ASIN, ASIND, ACOS, ACOSD, SINH, COSH, TANH	Real	Real

10.3.3 Intrinsic and Generic Function Usage

Figure 10-2 shows the use of intrinsic and generic function names. In this figure, a single executable program uses the name SIN in four distinct ways:

- As the name of a statement function
- As the name of a generic function
- As the name of a specific intrinsic function
- As the name of a user-defined function

Using the name in these four ways emphasizes the local and global properties of the name.

In Figure 10-2, the circled numbers are keyed to the notes that follow the figure.

```
C   COMPARE WAYS OF COMPUTING SINE.

PROGRAM SINES
REAL*Z8 X, PI
PARAMETER (PI=3.141592653589793238D0)
COMMON V(3)

C   Define SIN as a statement function ❶

SIN(X) = COS(PI/2-X)
DO 10 X = -PI, PI, 2*PI/100
CALL COMPUT(X)

C   Reference the statement function SIN ❷

10  WRITE (6,100) X, V, SIN(X)
100 FORMAT (5F10.7)
END

SUBROUTINE COMPUT(Y)
REAL*8 Y

C   Use intrinsic function SIN as actual argument ❸

INTRINSIC SIN
COMMON V(3)

C   Generic reference to double-precision sine ❹

V(1) = SIN(Y)
```

Figure 10-2: Multiple Function Name Usage

```

C   INTRINSIC FUNCTION SINE AS ACTUAL ARGUMENT ⑤
      CALL SUB(REAL(Y),SIN)
      END

      SUBROUTINE SUB(A,S)

C   Declare SIN as name of user function ⑥
      EXTERNAL SIN

C   Declare SIN as type REAL*8 ⑦
      REAL*8 SIN
      COMMON V(3)

C   Evaluate intrinsic function SIN ⑧
      V(2) = S(A)

C   Evaluate user-defined SIN function ⑨
      V(3) = SIN(A)
      END

C   Define the user SIN function ⑩
      REAL*8 FUNCTION SIN(X)
      INTEGER FACTOR
      SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)
      1 - X**7/FACTOR(7)
      END

      INTEGER FUNCTION FACTOR(N)
      FACTOR = 1
      DO 10 I=N,1,-1
10  FACTOR = FACTOR * I
      END

```

Figure 10-2 (Cont.): Multiple Function Name Usage

Notes:

- ① A statement function named SIN is defined in terms of the generic function name COS. Since the argument of COS is double precision, the double-precision cosine function will be evaluated. The statement function SIN is itself single precision.
- ② The statement function SIN is called.
- ③ The name SIN is declared intrinsic so that the single-precision intrinsic sine function can be passed as an actual argument at ⑤.
- ④ The generic function name SIN is used to refer to the double-precision sine function.
- ⑤ The single-precision intrinsic sine function is used as an actual argument.

Notes (Cont.)

- ⑥ The name SIN is declared a user-defined function name.
- ⑦ The type of SIN is declared double precision.
- ⑧ The single-precision sine function passed at ⑥ is evaluated.
- ⑨ The user-defined SIN function is evaluated.
- ⑩ The user-defined SIN function is defined as a simple Taylor series using a user-defined function FACTOR to compute the factorial function.

10.3.4 Character and Lexical Comparison Library Functions

Character library functions are functions that take character arguments and return integer, ASCII, or character values; lexical comparison library functions are functions that take character arguments and return logical values.

10.3.4.1 Character Functions

FORTRAN provides four character functions: LEN, INDEX, ICHAR, and CHAR.

LEN Function

The LEN function returns the length of a character expression. The LEN function has the form:

LEN(c)

where:

c
is a character expression. The value returned indicates how many bytes there are in the expression.

INDEX Function

The INDEX function searches for a substring (c2) in a specified character string (c1), and, if it finds the substring, returns the substring's starting position. If c2 occurs more than once in c1, the starting position of the first (leftmost) occurrence is returned. If c2 does not occur in c1, the value zero is returned. The INDEX function has the form:

INDEX(c1,c2)

where:

c1

is a character expression specifying the string to be searched for the substring specified by c2.

c2

is a character expression specifying the substring for which the starting location is to be determined.

ICHAR Function

The ICHAR function converts a character expression to its equivalent ASCII code and returns the ASCII value. ICHAR has the form:

ICHAR (c)

where:

c

is the character to be converted to an ASCII code. If c is longer than one byte, only the value of the first byte is returned; the remainder is ignored.

CHAR Function

The CHAR function converts an ASCII integer value to a character value and returns the character value. CHAR has the form:

CHAR (i)

where:

i

is an integer expression.

Examples

Examples illustrating the LEN and INDEX functions follow.

LEN Function Example:

```
SUBROUTINE REVERSE(S)
  CHARACTER T, S*(*)

  J = LEN(S)
  DO 10 I=1,J/2
    T = S(I:I)
    S(I:I) = S(J:J)
    S(J:J) = T
    J = J - 1
10  CONTINUE

  RETURN
  END
```

INDEX Function Example:

```
      SUBROUTINE FIND_SUBSTRINGS(SUB,S)
      CHARACTER*(*) SUB,S
      CHARACTER*132 MARKS

      I = 1
      MARKS = ' '

10     J = INDEX(S(I:),SUB)
      IF (J,NE,0) THEN
          I = I + (J-1)
          MARKS(I:I) = '#'
          I = I + 1
          IF (I,LE,LEN(S)) GO TO 10
      END IF

      WRITE (6,91) S, MARKS
91     FORMAT (2(/1X,A))
      END
```

10.3.4.2 Lexical Comparison Functions

The four lexical comparison functions provided with FORTRAN are:

- LLT, where LLT(X,Y) is equivalent to (X .LT. Y)
- LLE, where LLE(X,Y) is equivalent to (X .LE. Y)
- LGT, where LGT(X,Y) is equivalent to (X .GT. Y)
- LGE, where LGE(X,Y) is equivalent to (X .GE. Y)

The lexical functions have the form:

func(c,c)

where:

func

is one of the symbolic names: LLT, LLE, LGT, or LGE.

c

is a character expression.

The lexical comparison functions defined by the FORTRAN-77 standard are guaranteed to make comparisons according to the ASCII collating sequence, even on non-ASCII processors. On VAX systems, the lexical comparison functions are identical to the corresponding character relationals.

An example of the use of the lexical comparison functions is:

```
CHARACTER*10 CH2  
IF (LGT(CH2,'SMITH')) STOP
```

The IF statement in this example is equivalent to the following:

```
IF (CH2 .GT. 'SMITH') STOP
```


Chapter 11

VAX FORTRAN Input/Output

This chapter describes FORTRAN input/output (I/O) as implemented for VAX FORTRAN and provides information about FORTRAN I/O in relation to the VAX Record Management Services (RMS) and the VAX Run-Time Library (RTL).

The topics covered include:

- Overview of FORTRAN I/O (Section 11.1)
- Elements of I/O Processing (Section 11.2)
- Components of I/O statements (Section 11.3)
- READ statements (Section 11.4)
- WRITE statements (Section 11.5)
- REWRITE statement (Section 11.6)
- ACCEPT statement (Section 11.7)
- TYPE and PRINT statements (Section 11.8)

The statements listed here initiate data transfer operations. Other FORTRAN statements, which influence I/O processing but do not directly initiate data transfers, are described in Chapter 12, FORMAT Statements, and in Chapter 13, Auxiliary Input/Output Statements.

11.1 Overview of VAX FORTRAN I/O

This section introduces the concept of logical units, briefly describes the scope of interprocess communications, and lists and describes the different types of I/O statements and the optional forms of I/O statements.

11.1.1 Identifying Logical Input/Output Units

Logical unit numbers are integers from 0 to 99. For example:

```
READ (2,100) I,X,Y
```

This READ statement specifies that data is to be entered from the device or file corresponding to logical unit 2, in the format specified by the FORMAT statement labeled 100.

The association between the logical unit number and the physical device or file occurs at execution time. If necessary, you can change this association at execution time to match the needs of the program and the available resources. You do not need to change the logical unit numbers specified in the program. FORTRAN programs, therefore, are inherently device independent.

READ, WRITE, and REWRITE statements refer explicitly to a logical unit from or to which data is to be transferred. The logical unit can be connected to a device or file by means of an OPEN statement (see Section 13.1).

ACCEPT, TYPE, and PRINT statements do not refer explicitly to a logical unit (a file or device) from or to which data is to be transferred; they refer implicitly to a default logical unit. The ACCEPT statement is normally connected to the default input device, and the TYPE and PRINT statements are normally connected to the default output device. These defaults can be overridden with appropriate logical name assignments (see Section 13.3.1.1).

11.1.2 Types of I/O Statements

The type of an I/O statement depends on the organization of the file being accessed. The various types of I/O are as follows:

- Sequential I/O—transfers records sequentially to or from files or I/O devices such as terminals.
- Direct Access I/O—transfers records, selected by record number, to and from direct access files.
- Keyed I/O—transfers records, based on data values (keys) contained in the records, to and from indexed files.
- Internal I/O—transfers data between variables and arrays defined within a program.

11.1.3 Interprocess Communication

You can use standard FORTRAN I/O statements to communicate between processes on either the same computer or different computers.

- Mailboxes permit interprocess communication on the same computer.
- DECnet network facilities are used for interprocess communication on different computers. DECnet can also be used to process files on different computers.

Information on the preceding types of operations is provided in the *VAX FORTRAN User's Guide*.

11.1.4 Forms of I/O Statements

Each type of I/O statement can be coded in a variety of forms. The form you select depends on the nature of your data and how you want it treated. The I/O statement forms are *formatted*, *unformatted*, *list-directed*, and *namelist-directed*.

- Formatted I/O statements contain explicit format specifiers that are used to control the translation of data from internal (binary) form within a program to external (readable character) form in the records, or vice versa.
- List-directed and namelist-directed I/O statements are similar to formatted statements in function. However, they use different mechanisms to control the translation of data: formatted I/O statements use explicit format specifiers, and list-directed and namelist-directed I/O statements use data types.
- Unformatted I/O statements do not contain format specifiers and therefore do not translate the data being transferred. Unformatted I/O is especially appropriate where the output data will subsequently be used as input data. Unformatted I/O saves execution time by eliminating the data translation process, preserves greater precision in the external data, and usually conserves file storage space.

I/O statements transfer all data as records; that is, that which is read or written is a record. The amount of data that one of these records can contain depends on whether unformatted or formatted I/O is used to transfer the data. With unformatted I/O, the I/O statement alone specifies the amount of data to be transferred; with formatted I/O, the I/O statement and its associated format specifier jointly determine the amount of data to be transferred.

Normally, the data transferred by an I/O statement is read from or written to a single record. It is possible, however, for formatted, list-directed, and namelist-directed I/O statements to transfer data from or to more than one record.

Table 11-1 shows the various I/O statements, by category, that can be used in FORTRAN programs.

Table 11-1: Available I/O Statements

Statement Category	Statement Name					
	READ	WRITE	REWRITE	ACCEPT	TYPE	PRINT
Sequential						
Formatted	Yes	Yes	No	Yes	Yes	Yes
List-Directed	Yes	Yes	No	Yes	Yes	Yes
Namelist-Directed	Yes	Yes	No	Yes	Yes	Yes
Unformatted	Yes	Yes	No	No	No	No
Direct						
Formatted	Yes	Yes	No	No	No	No
Unformatted	Yes	Yes	No	No	No	No
Indexed						
Formatted	Yes	Yes	Yes	No	No	No
Unformatted	Yes	Yes	Yes	No	No	No
Internal						
Formatted	Yes	Yes	No	No	No	No
List-Directed	Yes	Yes	No	No	No	No
Unformatted	No	No	No	No	No	No

11.2 Elements of I/O Processing

The following sections describe in general terms the elements of FORTRAN I/O processing. The topics covered include:

- VAX/VMS file specifications (Section 11.2.1)
- Logical names, as used in FORTRAN, and logical unit numbers (Section 11.2.2)
- FORTRAN file organizations, I/O record formats, and access modes (Section 11.2.3)

11.2.1 File Specifications

VAX/VMS file specifications are described in detail in Chapter 1. The discussion of file specifications is abbreviated in this section, concentrating on how to identify files in I/O statements.

A complete VAX/VMS file specification has the form:

```
node::device:[directory]filename.filetype;version
```

For example:

```
BOSTON::USERD:[SMITH]TEST.DAT;2
```

You can associate a file specification with a logical unit by using a logical name assignment (see Section 11.2.2) or by using the OPEN statement (see Section 11.2.3). If you do not specify such an association or if you omit elements of the file specification, the system supplies default values, as follows:

- If you omit the node, the local computer is used.
- If you omit the device or directory, the current user default is used.
- If you omit the file name, the system supplies FOR0nn, where nn is the logical unit number.
- If you omit the file type, the system supplies DAT.
- If you omit the version number, the system supplies either the highest current version number (for input) or the highest current version number plus 1 (for output).

For example, if your default device is USERD and your default directory is SMITH, and you specify:

```
READ (8,100)
  .
  .
  .
WRITE (9,200)
```

The default input and output file specifications are respectively:

```
USERD:[SMITH]FOR008.DAT;n
```

and

```
USERD:[SMITH]FOR009.DAT;m
```

Where n equals the highest current version number of FOR008.DAT and m is 1 greater than the highest existing version number of FOR009.DAT.

11.2.2 Logical Names and Logical Unit Numbers

You can use the logical name mechanism of the VMS operating system to associate logical units with file specifications. A logical name is an alphanumeric string, up to 63 characters long, that you can use instead of a file specification.

The operating system supplies a number of predefined logical names that are already associated with particular file specifications. Table 11-2 lists the logical names of special interest to FORTRAN users. FORTRAN logical unit names are shown in Table 11-3.

Table 11-2: Predefined System Logical Names

Name	Meaning	Default
SYS\$COMMAND	Default command stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch job input command file.
SYS\$DISK	Default disk device	As specified by the user.
SYS\$ERROR	Default error stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch job log file.
SYS\$INPUT	Default input stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch command file.
SYS\$OUTPUT	Default output stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch log file.

You can dynamically create a logical name and associate it with a file specification by means of the VAX/VMS ASSIGN command. For example, before program execution, you can associate the logical names in your program with the file specification appropriate to your needs.

For example:

```
* ASSIGN USERD:[SMITH]TEST.DAT;2 LOGNAM
```

The preceding command creates the logical name LOGNAM and associates it with the file specification USERD:[SMITH]TEST.DAT;2. As a result, this file specification is used whenever the logical name LOGNAM is encountered during program execution.

Logical names provide great flexibility because they can be associated not only with a complete file specification, but with a portion of a file specification (that is, either a device or a device and a directory), or even another logical name.

11.2.2.1 FORTRAN Logical Names

Usually, FORTRAN I/O is performed by associating a logical unit number with a device or file. VAX/VMS logical names provide an additional level of association; a user-specified logical name can be associated with a logical unit number.

VAX FORTRAN provides predefined logical names in the form:

FOR0nn

where:

nn

corresponds to the logical unit number.

By default, each FORTRAN logical name is associated with a file named FOR0nn.DAT, which is assumed to be located on your default disk under your default directory. For example:

```
WRITE (17,200)
```

If you enter the preceding statement without including an explicit file specification, the data is written to a file named FOR017.DAT on your default disk under your default directory.

You can change the file specification associated with a FORTRAN logical unit number by using the ASSIGN command to change the file associated with the corresponding FORTRAN logical name. For example:

```
$ ASSIGN USERD:[SMITH]TEST.DAT;2 FOR017
```

The preceding command associates the FORTRAN logical name FOR017 (and therefore logical unit 17) with file TEST.DAT;2 on device USERD in directory [SMITH].

You can also associate the FORTRAN logical names with any of the predefined system logical names, as shown in the following two examples:

1. The following command associates logical unit 5 with the default input device, for example, the batch input stream.

```
$ ASSIGN SYS$INPUT FOR005
```

2. The following command associates logical unit 6 with the default output device, for example, the batch output stream.

```
$ ASSIGN SYS$OUTPUT FOR006
```

VAX FORTRAN provides default logical name assignments for logical units 5 and 6, as shown in the preceding examples.

11.2.2.2 Implied FORTRAN Logical Unit Numbers

The ACCEPT, PRINT, and TYPE statements, and optionally the READ and WRITE statements, do not include an explicit logical unit number. Each of these FORTRAN statements uses an implicit logical unit number and logical name. Each logical name is, in turn, associated by default with one of the system's predefined logical names. Table 11-3 shows these relationships.

Table 11-3: Implicit FORTRAN Logical Units

Statement	FORTRAN Logical Name	System Logical Name
READ (*,f) list	FOR\$READ	SY\$INPUT
READ f,list	FOR\$READ	SY\$\$INPUT
ACCEPT f,list	FOR\$ACCEPT	SY\$INPUT
WRITE (*,f) list	FOR\$PRINT	SY\$OUTPUT
PRINT f,list	FOR\$PRINT	SY\$OUTPUT
TYPE f,list	FOR\$TYPE	SY\$OUTPUT

You can change the file specifications associated with these FORTRAN logical names, as you would any other FORTRAN logical name, by means of the VMS ASSIGN command. For example:

```
# ASSIGN USERD:[SMITH]TEST.DAT;2 FOR$READ
```

Following execution of the preceding command, the READ statement's logical name (FOR\$READ) refers to the file TEST.DAT;2 on device USERD in directory [SMITH].

11.2.2.3 File Specification in the OPEN Statement

You can use the FILE and DEFAULTFILE keywords of the OPEN statement to specify the complete definition of a particular file to be opened on a logical unit. (Section 13.1 describes the OPEN statement in greater detail.) For example:

```
OPEN (UNIT=4, FILE='USERD:[SMITH]TEST.DAT;2', STATUS='OLD')
```

In the preceding example, the file TEST.DAT;2 on device USERD in directory SMITH is to be opened on logical unit 4. Neither the default file specification (FOR004.DAT) nor the FORTRAN logical name FOR004 is used. The value of the FILE keyword can be a character constant, variable, or expression.

In the following interactive example, the file name is supplied by the user and the DEFAULTFILE keyword supplies the default values for the file specification string.

```
CHARACTER*9 DOC  
TYPE *, 'ENTER FILE NAME (WITHIN APOSTROPHES)'  
ACCEPT *, DOC  
OPEN (UNIT=2, FILE=DOC,  
1   DEFAULTFILE='USERD:[ARCHIVE].TXT',  
1   STATUS='OLD')
```

In the preceding example, the file that is to be opened is located on device USERD in directory ARCHIVE, with the file name supplied in DOC and the file type TXT. The DEFAULTFILE specification overrides your process default device and directory.

You can also specify a logical name as the value of the FILE keyword, if the logical name is associated with a file specification. For example:

```
$ ASSIGN USERD:[SMITH]TEST.DAT LOGNAM
```

The preceding command assigns the logical name LOGNAM to the file specification USERD:[SMITH]TEST.DAT. The logical name can then be used in an OPEN statement, as follows:

```
OPEN (UNIT=19, FILE='LOGNAM', STATUS='OLD')
```

When an I/O statement refers to logical unit 19, the system uses the file specification associated with logical name LOGNAM.

If the value specified for the FILE keyword has no associated file specification, it is regarded as a true file name rather than as a logical name. That is, if LOGNAM had not been previously associated with the file specification USERD:[SMITH]TEST.DAT by means of an ASSIGN command, then the above OPEN statement indicates that a file named LOGNAM.DAT is located on the default device, in the default directory.

A logical name specified in an OPEN statement must not contain brackets, semicolons, or periods. The system treats any name containing these punctuation marks as a file specification, not as a logical name.

11.2.2.4 Assigning Files to Logical Units—Summary

As described in the preceding sections, you can assign files to logical units in any of three ways:

- By using default logical names. In the following example, the READ statement causes the logical unit FOR007 to be associated with the file FOR007.DAT by default, and the TYPE statement causes the logical unit FOR\$TYPE to be associated with SYS\$OUTPUT by default.

```
READ (7,100)
  .
  .
  .
TYPE 100
```

- By specifying a logical name in an OPEN statement. For example:

```
OPEN (UNIT=7, FILE='LOGNAM', STATUS='OLD')
```

- By supplying a file specification in an OPEN statement. For example:

```
OPEN (UNIT=7, FILE='FILNAM.DAT', STATUS='OLD')
```

You can use the ASSIGN command to change the association of logical names and file specifications.

A logical name used with the FILE keyword of the OPEN statement must be associated with a file specification, and the character expression specified for the FILE keyword must contain no punctuation marks. Otherwise, the logical name is treated as a true file specification.

You use the VAX/VMS SHOW LOGICAL command to determine the current associations of logical names and file specifications.

To remove the association of a logical name and a file specification, use the DEASSIGN command, in the form:

```
$ DEASSIGN logical-name
```

11.2.3 File Organizations, I/O Record Formats, and Access Modes

A distinction must be made between the way in which files are organized and the way in which records are accessed. The term “file organization” applies to the way records are physically arranged on a storage device. “Record access” refers to the method used to read records from or write records to a file, regardless of its organization. A file’s organization is specified when the file is created, and cannot be changed. In contrast, record access is specified each time the file is opened, and can be different each time.

The following sections describe in general terms the elements of FORTRAN I/O processing: files, internal files, I/O records, and access modes.

11.2.3.1 File Organizations

A file is a collection of logically related records that are arranged in a specific order and treated as a unit. The arrangement or organization of a file is determined when the file is created.

VAX FORTRAN supports three kinds of file organization: sequential, relative, and indexed. The organization of a file is specified by means of the ORGANIZATION keyword in the OPEN statement, as described in Section 13.1.19.

Files are normally stored on disk. Sequential files, however, can be stored on either magnetic tape or disk. Other peripheral devices, such as terminals, card readers, and line printers, are treated as sequential files.

The three kinds of file organization are discussed individually under the headings that follow.

Sequential Organization

A sequentially organized file consists of records arranged in the sequence in which they are written to the file (the first record written is the first record in the file, the second record written is the second record in the file, and so on). As a result, records can be added only at the end of the file.

Sequential file organization is permitted on all devices supported by the VMS operating system.

Relative Organization

A relative file consists of numbered positions, called cells. These cells are of fixed equal length and are consecutively numbered from 1 to n, where 1 is the first cell, and n is the last available cell in the file. Each cell either contains a single record or is empty.

Records in a relative file are accessed according to cell number. A cell number is a record's relative record number, that is, its location relative to the beginning of the file. By specifying relative record numbers, you can directly retrieve, add, or delete records regardless of their locations.

Relative files are supported only on disk devices.

Indexed Organization

An indexed file consists of two or more separate sections: one section contains the data records, and the other(s) contain the index(es). When an indexed file is created, each index is associated with a specification defining a field, called a key field, within each record. A record in an indexed file must contain at least one key. This mandatory key, called the primary key, determines the location of the records within the body of the file.

The keys of all records are collected to form one or more structured indexes, through which records are always accessed. The structure of the index(es) allows a program to access records in an indexed file either randomly, by specifying particular key values, or sequentially, by retrieving records with increasing key values. In addition, keyed access and sequential access can be mixed. The term Indexed Sequential Access Method (ISAM) refers to this dynamic access feature.

Indexed files are supported only on disk devices. See Chapter 15 for more information on indexed files.

11.2.3.2 Internal Files

An internal file is designated internal storage space that is manipulated to facilitate internal I/O. Its use with formatted and list-directed sequential READ and WRITE statements eliminates the need to use the ENCODE and DECODE statements for internal I/O (see Appendix A).

An internal file consists of a character variable, a character array element, a character array, or a character substring; a record in an internal file consists of any of these data items except a character array.

If an internal file is made up of a single character variable, array element, or substring, that file comprises a single record whose length is the same as the length of the variable, array element, or substring. If an internal file is made up of a character array, that file comprises a sequence of records, with each record consisting of a single array element. The sequence of records in an internal file is determined by the order of subscript progression.

A record in an internal file can be read only if the character variable, array element, or substring comprising the record has been defined; that is, a value has been assigned to the record.

Prior to data transfer, an internal file is always positioned at the beginning of the first record.

11.2.3.3 I/O Record Formats

An I/O record is a collection of data items, called fields, that are logically related and are processed as a unit.

NOTE

I/O records are not to be confused with record entities declared in a program as structured data items. There is no relationship between structured data items and I/O records. Structured data items are described in Section 6.2.5.

Generally, each FORTRAN I/O statement transfers one record. The exceptions are formatted, list-directed, and namelist-directed I/O statements, which can transfer additional records.

If an input statement does not use all of the data fields in a record, the remaining fields are ignored. If an input statement requires more data fields than the record contains, either an error condition occurs or, in the case of formatted input, all fields are read as spaces.

If an output statement attempts to write more data fields than the record can contain, an error condition occurs. If an output statement transfers less data than is required to fill a fixed-length record, the record is filled with spaces (if it is a formatted record) or zeros (if it is an unformatted record).

Records are stored in one of four formats:

- Fixed-length
- Variable-length
- Segmented
- Stream

Fixed-length and variable-length formats can be used with sequential, relative, or indexed file organization. Segmented format is unique to FORTRAN; it is not used by other VMS-supported languages. It can only be used with sequential file organization, and only for unformatted sequential access. You should not use segmented records for files that are read by programs written in languages other than FORTRAN. Stream format can only be used with sequential file organization.

The various kinds of I/O record formats are discussed individually under the headings that follow.

Fixed-Length Records

When you specify fixed-length records (see Section 13.1.23), you are specifying that all records in the file contain the same number of bytes. When you create a file that is to contain fixed-length records, you must specify the record size (see Section 13.1.21). A sequentially organized file opened for direct access must contain fixed-length records, to allow the record number to be computed correctly.

Variable-Length Records

Variable-length records can contain any number of bytes, up to a specified maximum. These records are prefixed by a count field, indicating the number of bytes in the record. The count field comprises two bytes on a disk device and four bytes on magnetic tape. The value stored in the count field indicates the number of data bytes in the record. Variable-length records in relative files are actually stored in fixed-length cells, the size of which must be specified by means of the RECL keyword of the OPEN statement (see Section 13.1.21). This RECL value specifies the largest record that can be stored in the file.

The count field of a variable-length record is available when you read the record by issuing a READ statement with a Q format descriptor. You can then use the count field information to determine how many bytes should be in an I/O list.

Segmented Records

A segmented record is a single logical record consisting of one or more variable-length, unformatted records in a sequentially organized file. Each variable-length record constitutes a segment. The length of a segmented record is arbitrary. Segmented records are useful when you want to write exceptionally long records but cannot or do not wish to define one long variable-length record. Unformatted data written to sequentially organized files using sequential access is stored as segmented records by default.

Because there is no limit on the size of a segmented record, each variable-length record in the segmented record contains control information to indicate that it is one of the following:

- The first segment
- The last segment
- The only segment
- None of the above

This control information is contained in the first two bytes of each segment of a segmented record. Therefore, when you wish to access an unformatted sequential file that contains variable-length records, you must specify RECORDTYPE='VARIABLE' when you open the file. Otherwise, the first two bytes of each record will be mistakenly interpreted as control information, and errors will probably result.

Stream Records

A stream-type record is a variable-length record whose length is indicated by explicit record terminators embedded in the data, not by a count. These terminators are automatically added when you write records to a stream-type file and are removed when you read records.

There are three varieties of stream-type files, each using a different record terminator:

- STREAM files use the 2-character sequence consisting of a carriage-return and a line-feed as the record terminator.
- STREAM_CR files use only a carriage-return as the terminator.
- STREAM_LF files use only a line-feed as the terminator.

11.2.3.4 Record Access Modes

Access mode is the method a program uses to retrieve and store records in a file. The access mode is specified as part of each I/O statement. VAX FORTRAN supports three record access modes:

- Sequential
- Direct
- Keyed

Your choice of record access mode is affected by the organization of the file to be accessed. For example, the sequential access mode can be used with sequential, relative, and indexed files; but the keyed access mode can be used only with indexed organization files.

Table 11-4 shows all the valid combinations of access mode and file organization.

Table 11-4: Valid Combinations of Record Access Mode

File Organization	Access Mode		
	Sequential	Direct	Keyed
Sequential	Yes	Yes ¹	No
Relative	Yes	Yes	No
Indexed	Yes	No	Yes

¹ Fixed-length records only.

The three kinds of access mode are discussed individually under the headings that follow.

Sequential Access Mode

If you select sequential access mode for files with sequential or relative organization, records are written to or read from the file starting at the beginning and continuing through the file, one record after another. For files with indexed organization, sequential access can be used to read or write all records according to ascending key values. Sequential access to indexed files can also be used with keyed access to read or write a group of records at a specified point in the file.

When you use sequential access for files with sequential and relative organization, a particular record can be retrieved only after all the records preceding it have been read.

Writing records by means of sequential access also varies according to the file organization.

- For a file with sequential organization, new records can be written only at the end of the file.
- For a file with relative organization, a new record can be written at any point, replacing the existing record in the specified cell. For example, if two records are read from a relative file and then a record is written, the new record occupies cell 3 of the file.
- For a file with indexed organization, records can be written in any order, and READ operations refer to the next record with the same or next higher specified key value.

Direct Access Mode

If you select direct access mode, you determine the order in which records are read or written. Each READ or WRITE statement must include the relative record number, indicating the record to be read or written.

You can access relative files directly. You can also access a sequential disk file directly if it contains fixed-length records. Because direct access uses cell numbers to find records, you can issue successive READ or WRITE statements requesting records that either precede or follow previously requested records. For example, the following statements, appearing in a program in the order shown here, read record 24 and then read record 10.

```
READ (12,REC=24) I  
READ (12,REC=10) J
```

Keyed Access Mode

If you select keyed access mode, you determine the order in which records are read or written by means of character values or integer values called keys. Each READ statement contains the key that locates the record. The key value in the I/O statement is compared with index entries until the record is located.

When you insert a new record, the values contained in the key fields of the record determine the record's placement in the file; you do not have to indicate a key.

You can use keyed access only for files with an indexed organization.

Your program can mix keyed access and sequential access I/O statements on the same file. You can use keyed I/O statements to position the file to a particular record and then use sequential I/O statements to access records with increasing key values in the current key-of-reference.

11.3 Components of I/O Statements

I/O statements are composed of three basic components: the *statement keyword*, the *control list*, and the *I/O list*.

The six statement keywords that represent input and output operations are:

Input Operations	Output Operations
READ	WRITE
ACCEPT	REWRITE
	TYPE
	PRINT

These statements are fully described in Sections 11.4 through 11.8.

The control list and the I/O list are discussed in Sections 11.3.1 and 11.3.2, respectively.

11.3.1 Control List

The control list of an I/O statement is a list of one or more parameters that specify the following:

- The logical unit to be acted upon
- The internal file to be acted upon
- Whether formatting is to be used for data editing, and, if it is, the format specification
- The NAMELIST group-name specification
- The cell number of a direct access record to be accessed
- The key and key-of-reference of a keyed access record to be accessed
- The name of a variable to contain the completion status of an I/O operation
- The label of a statement to which control is transferred in the event of an error or end-of-file condition

The type of a statement can always be determined by the contents of its control list. For example, the control list of a formatted I/O statement always contains a format specifier (FMT=f or f), whereas that of a list-directed I/O statement always contains an asterisk in place of a format specifier.

The control list has the form:

(p[,p]...)

where:

p

is of the form:

[keyword =] value

The keywords and values are explained in the following sections.

11.3.1.1 Logical Unit Specifier

The logical unit specifier is a parameter that specifies the logical unit to be accessed.

The logical unit specifier has one of the following forms:

```
[UNIT=]u  
[UNIT=]*
```

where:

u

is an integer expression with a value in the range 0 through 99 that refers to a specific file or I/O device. If necessary, the value is converted to integer data type before being used.

specifies that the default input or output unit is to be accessed.

The keyword UNIT is optional only if the logical unit specifier is the first parameter in the control list.

A logical unit number is assigned to a file or device in one of two ways:

- Explicitly through an OPEN statement (see Section 13.1)
- Implicitly by the system (see Section 11.2.2.2)

11.3.1.2 Internal File Specifier

An internal file specifier is a parameter that specifies the internal file to be used.

The internal file specifier has the form:

```
[UNIT=]cv
```

where:

cv

is a character scalar memory reference or a character array name reference.

The external logical unit specifier and the internal file specifier are mutually exclusive. The keyword UNIT is optional if the internal file specifier is the first parameter in the control list.

See Section 11.2.3.2 for more information on internal files.

11.3.1.3 Format Specifiers

The format specifier is a parameter that specifies that explicit or list-directed formatting is to be used and, in the case of explicit formatting, identifies the parameter that will control the formatting.

The format specifier has the forms:

[FMT=]f
[FMT=]*

where:

f
is the statement label of a FORMAT statement, an integer variable that has been assigned a FORMAT statement label with an ASSIGN statement, or the name of an array, array element, or character expression containing a run-time format.

specifies list-directed formatting.

The keyword FMT is optional only if the format specifier is the second parameter in the control list, and the first parameter is a logical unit or internal file specifier without the optional keyword UNIT.

Chapter 12 describes FORMAT statements. Section 12.7 describes the interaction between formats and I/O statements.

In sequential I/O statements, you can use an asterisk instead of a format specifier to denote list-directed formatting. See Sections 11.4.1.2 and 11.5.1.2.

11.3.1.4 Namelist Specifier

The namelist specifier is a parameter that specifies that namelist-directed I/O is being used and identifies the group-name of the list of entities that may be modified on input or written on output.

The namelist specifier has the form:

[NML=]group-name

where:

group-name

is the name of a list previously defined in a NAMELIST statement.

The keyword NML is optional only if (1) the namelist specifier is the second parameter in the control list and (2) the first parameter is a logical unit specifier without an optional keyword UNIT. A namelist specifier cannot be used in a statement that contains a format specifier.

11.3.1.5 Record Specifier

The record specifier is a parameter that specifies the number of the record to be accessed in a file with relative organization.

The record specifier has the forms:

$$\left\{ \text{REC} = r \right\}$$

where:

r

is a numeric expression with a value that represents the position in a direct access file of the record to be accessed. The value must be greater than or equal to one, and less than or equal to the maximum number of record cells allowed in the file. If necessary, a record number is converted to integer data type before being used.

11.3.1.6 Key-Field-Value Specifier

The key-field-value specifier is a parameter that specifies the key field value of a record to be accessed in an indexed file. Indexed files are composed of records that have one or more fields in common; that is, the byte offset, type, and length of the field(s) are the same in each record in any given indexed file.

The key-field-value specifier has two components:

- An expression, which specifies the key field value to be used in locating the record to be transferred
- A match criterion, which specifies the selection conditions

A key-field-value specifier has one of the following forms:

```
KEY=val  
KEYEQ=val  
KEYGE=val  
KEYGT=val
```

where:

val

is a character expression or an integer expression. Character expressions must be used with character key fields, and integer expressions must be used with integer key fields.

An integer expression in a key-field-value specifier cannot contain real and complex values.

A character expression in a key-field-value specifier can be an ASCII string in one of the following forms:

- A character expression
- A BYTE (LOGICAL*1) array name containing Hollerith data

The length of the character expression is the length of the character key field value or the length of the BYTE array. If the length of the expression is greater than the length of the key field, an error occurs. If the length of the expression is less than the length of the key field, a generic key value search rather than an exact key field value search is made.

The match criterion specifies which key values in the record can match the expression. There are three possible criteria:

- Equal. The key field value must be equal to the expression specified.
- Greater. The key field value must be greater than the expression specified.
- Greater than or equal. The key field value must be greater than or equal to the expression specified.

The following parameters are used to establish the desired match criterion:

KEY=val	}	specifies an equal match.
KEYEQ=val		
KEYGT=val		specifies a greater than match.
KEYGE=val		specifies a greater than or equal match.

For character expressions, the comparison is made according to the ASCII collating sequence.

For integer expressions, the comparison is made according to the signed integer value.

For character keys, either generic match or exact match can be used. Generic match applies if the expression in the I/O statement's key specifier is shorter than the key field in the record. In this case, only the leftmost characters of the key field are used for the match.

For example, if the expression is 'ABCD' and the key field is 10 characters long, then an equal match is obtained for the first record containing 'ABCD' as the first 4 bytes of the key field. The remaining six characters are arbitrary.

An approximate-generic match occurs when approximate match (KEYGT or KEYGE) is selected in addition to generic match. In that case, only the leftmost characters are used for comparison.

For example, if the expression is 'ABCD', the key field is five characters long, and a greater-than match is selected, then the value 'ABCDA' does not match; 'ABCEX' does match.

11.3.1.7 Key-of-Reference Specifier

The key-of-reference specifier may optionally be included with a key-field-value specifier; it is used to specify the key field index that is to be searched for the specified key field value.

The key-of-reference specifier has the form:

KEYID=kn

where:

kn

is an integer expression, called the key-of-reference number, that designates the key field index to be searched.

The key-of-reference number is an integer value in the range zero to the maximum key number defined for the file. A value of zero specifies the primary key, a value of one specifies the first alternate key, and so forth.

If no key-of-reference number is given, it defaults to the last specification given in a keyed I/O statement for that logical unit.

11.3.1.8 I/O Status Specifier

The I/O status specifier designates a variable in which a value is stored that indicates whether an error or end-of-file condition exists. If the value is zero, no error or end-of-file condition exists. If the value is positive, an error condition exists. If the value is negative, an end-of-file condition exists, but an error condition does not.

The I/O status specifier has the form:

IOSTAT=ios

where:

ios

is an integer scalar memory reference.

Refer to Section 18.3 for more information on the error numbers returned by IOSTAT.

11.3.1.9 Transfer-of-Control Specifiers

The transfer-of-control specifiers are parameters that transfer control of the program to a specific statement in the event of an end-of-file condition or an error condition.

The transfer-of-control specifiers have the form:

END=s

ERR=s

where:

s

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

A sequential READ statement can include either or both of the above specifications, in any order. WRITE, REWRITE, direct access READ, and keyed access READ statements can include only the ERR=s specification.

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

An end-of-file condition occurs when no more records exist in a file during a sequential read, or when an end-file record produced by the ENDFILE statement is encountered (see Section 9.6). End-of-file conditions do not occur in direct access or keyed access READ statements. If a READ statement encounters an end-of-file condition during an I/O operation, it transfers control to the statement named in the END=s specification; if there is no END=s specification and no IOSTAT specifier, an error occurs.

If a READ, WRITE, or REWRITE statement encounters an error condition during an I/O operation, it transfers control to the statement whose label appears in the ERR=s specification. If neither an ERR specifier nor an IOSTAT specifier is present, the I/O error terminates program execution.

The *VAX FORTRAN User's Guide* describes system subroutines that you can use to control error processing. To obtain information from the I/O system on the type of error that occurred, use the IOSTAT parameter discussed in Section 11.3.1.8.

Examples of I/O statements follow:

1. The following READ statement transfers control to statement 550 if an end-of-file condition occurs on logical unit 8.

```
READ (8,END=550) (MATRIX(K),K=1,100)
```

2. The following WRITE statement transfers control to statement 390 if an error occurs while it is being executed.

```
WRITE (6,50,ERR=390) VAR1, VAR2, VAR3
```

3. The following READ statement transfers control to statement 150 (if an error occurs while it is being executed) or to statement 200 (if an end-of-file condition occurs).

```
READ (1,FORM,ERR=150,END=200) ARRAY
```

11.3.1.10 Rules for Specifying Control List Parameters—Summary

The FORTRAN I/O statements described in Sections 11.4 through 11.8 are subject to the following syntactical rules:

1. For the keyword form of control-list parameters (that is, when the control-list parameter is specified with a keyword and equal sign):
 - The control-list parameters can appear in any order in the control list. An exception to this rule occurs when keyword and nonkeyword forms are intermixed in the same I/O statement. In this case, the provisions of the rules for nonkeyword form take precedence.

2. For the nonkeyword form of control-list parameters (that is, when the control-list parameter is specified without a keyword and equal sign):
 - Either the logical unit specifier or the internal file specifier must occupy the first (leftmost) position in the control list.
 - When used with a logical unit specifier or internal file specifier, the nonkeyword form of the format or namelist specifier must occupy the second position in the control list; the unit or internal file specifier must also be in nonkeyword form (and therefore occupy the first position in the control list).
 - The nonkeyword form of the direct access record specifier must immediately follow the nonkeyword form of the logical unit specifier.

11.3.2 I/O List

The I/O list in an input or output statement contains the scalar references, array name references, and aggregate references specifying the memory locations from which or to which data will be transferred. (See Section 6.2.6 for a description of the different types of references.)

The I/O list in an input statement cannot contain constants and expressions because these do not specify named memory locations that can be referenced later in the program. The I/O list in an output statement can contain constants and expressions, however, because the compiler can use temporary memory locations to hold these values during the execution of the I/O statement.

An I/O list has the following form:

s[,s]...

where:

s

is a simple list element or an implied-DO list.

The I/O statement assigns values to, or transfers values from, the list elements in the order in which they appear, from left to right.

11.3.2.1 Simple List Elements

A simple I/O list element can be a scalar reference, scalar array name reference, or aggregate reference. For example:

```
WRITE (5,10) J, K(3), 4, (L+4)/2, N
```


When you use an array name reference or an aggregate reference in an I/O list, an input statement reads enough data to fill every element of the array or aggregate; an output statement writes all of the values in the array or aggregate. Data transfer begins with the initial element of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most rapidly. For example, the following defines a two-dimensional array:

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

If the name `ARRAY`, with no subscripts, appears in a `READ` statement, that statement assigns values from the input record(s) to `ARRAY(1,1)`, `ARRAY(2,1)`, `ARRAY(3,1)`, `ARRAY(1,2)`, and so on through `ARRAY(3,3)`.

In an input statement, variables in the I/O list can be used in array subscripts later in the list. For example:

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

The input record contains the following values: 1,3,721.73

When the `READ` statement is executed, the first input value is assigned to `J` and the second to `K`, thereby establishing the actual subscript values for `ARRAY(J,K)`. Then the value 721.73 is assigned to `ARRAY(1,3)`. Variables that are to be used as subscripts in this way must appear before (to the left of) their use as the array subscripts in the I/O list.

An output statement I/O list may contain any valid expression. However, this expression must not attempt any further I/O operations on the same logical unit. For example, an output statement I/O list expression must not refer to a function subprogram that performs I/O on the same logical unit.

An input statement I/O list must not contain a constant or an expression, except as a subscript expression in an array reference or as an expression in a substring reference.

Aggregate references can be used only in unformatted input and output statements. When multiple array names or aggregate references are used in the I/O list of an unformatted input or output statement, only one record is read or written regardless of how many array name references or aggregate references appear in the list.

11.3.2.2 Implied-DO Lists in I/O Statements

An implied-DO list is an I/O list element that functions as though it were a part of an I/O statement within a DO loop. Implied-DO lists can be used to:

- Specify iteration of part of an I/O list
- Transfer part of an array
- Transfer array elements in a sequence different from the order of subscript progression

An implied-DO list has the form:

```
(list, i=e1,e2[,e3])
```

where:

list

is an I/O list.

i

is an integer or real variable.

e1,e2,e3

are arithmetic expressions.

The variable *i* and the parameters *e1*, *e2*, and *e3* have the same forms and the same functions that they have in the DO statement (see Section 9.3). The list immediately preceding the DO loop parameter is the range of the implied-DO loop. Elements in that list can reference *i*, but they must not alter the value of *i*. Some examples of the use of implied-DO lists follow.

Examples

1. The following two WRITE statements have the same effect.

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

and

```
WRITE (3,200) A,B,C,A,B,C,A,B,C
```

2. In the following example, the I/O list consists of an implied-DO list containing another implied-DO list nested within it.

```
WRITE (6) (I, (J,P(I),Q(I,J), J=1,L), I=1,M)
```

Together, the implied-DO lists write a total of $(1+3*L)*M$ fields, varying the Js for each value of I.

3. In a series of nested implied-DO lists, the parentheses indicate the nesting (see Section 9.3.1.2). Execution of the innermost lists is repeated most often. For example:

```
WRITE (6,150) ((FORM(K,L), L=1,10), K=1,10,2)
150 FORMAT (F10,2)
```

Because the inner DO loop is executed 10 times for each iteration of the outer loop, the second subscript, *L*, advances from 1 through 10 for each increment of the first subscript. This is the reverse of the order of subscript progression. In addition, *K* is incremented by 2; thus, only the odd-numbered rows of the array are output.

4. The entire list of an implied-DO list is transmitted before the control variable is incremented. For example:

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

In this example, *P*(1), *Q*(1,1), *Q*(1,2) ..., *Q*(1,10) are read before *I* is incremented to 2.

5. When processing multidimensional arrays, you can use a combination of fixed subscripts and subscripts that vary according to an implied-DO list. For example:

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

This statement assigns input values to BOX(1,1) through BOX(1,10) and then terminates without affecting other elements of the array.

6. The value of the control variable can also be output directly. For example:

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

This statement simply prints the integers 1 through 20.

If the I/O statement containing an implied-DO list terminates abnormally (that is, with an END= or ERR= transfer or with an IOSTAT value other than zero), the loop control variable becomes undefined.

11.4 READ Statements

The READ statements transfer input data to internal storage from records contained in external logical units or to internal storage from internal files.

The four types of READ statement—*sequential*, *direct*, *indexed*, and *internal*—are described in Sections 11.4.1 through 11.4.4.

11.4.1 Sequential READ Statements

Sequential READ statements transfer input data to internal storage from external records accessed under the sequential mode of access.

The formats of the four forms of sequential READ statement—*formatted*, *list-directed*, *namelist-directed*, and *unformatted*—are as follows:

Formatted

```
READ (extu,fmt[,iostat][,err][,end]) [iolist]
```

```
READ f[,iolist]
```

List-Directed

```
READ (extu,*[,iostat][,err][,end]) [iolist]
```

```
READ *[,iolist]
```

Namelist-Directed

```
READ (extu,nml[,iostat][,err][,end])
```

```
READ n
```

Unformatted

READ (extu[,iostat][,err][,end]) [iolist]

The meanings of the symbolic abbreviations used to represent control-list parameters in the preceding command lines are as follows:

- extu—a logical unit specifier.
- fmt—a format specifier.
- f—the nonkeyword form of a format specifier. See fmt, above.
- *—specifies list-directed formatting. You can also use FMT=*.
- nml—a namelist specifier.
- n—the nonkeyword form of a namelist specifier. See nml, above.
- iostat—an I/O status specifier.
- err/end—transfer-of-control specifiers.

The I/O-list parameter is represented by the symbolic abbreviation iolist.

All of the parameters used in I/O statements are described in Sections 11.3.1 (control-list parameters) and 11.3.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 11.3.1.10.

The uses and effects of the four forms of sequential READ statements are described in Sections 11.4.1.1 through 11.4.1.4.

11.4.1.1 Formatted Sequential READ Statement

The formatted sequential READ statement performs the following operations:

- Reads character data from one or more external records accessed under the sequential or keyed mode of access.
- Translates the data from character to binary form using format specifications to provide editing.
- Assigns the translated data to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list.

If the number of I/O list elements in a statement is less than the number of fields in an input record, the statement ignores the excess fields.

See Section 11.4.3 for information about the combined use of formatted sequential READ statements and indexed READ statements under the keyed mode of access.

11.4.1.2 List-Directed Sequential READ Statement

The list-directed sequential READ statement performs the following operations:

- Reads character data from records accessed under the sequential mode of access.
- Translates the data from external to binary form using the data types of the elements in the I/O list, and the forms of the data, to provide editing.
- Assigns the translated data to the elements in the I/O list in the order, from left to right, in which those elements appear in the list.

The external records from which list-directed READ statements read data contain a sequence of values and value separators.

A value in one of these records may be any one of the following:

- A *constant*—each constant has the form of the corresponding FORTRAN constant. Input constants can be any of the following data types: integer, real, logical, complex, and character. The data type of the constant determines the data type of the value and the translation from external to internal form.

A numeric list element can correspond only to a numeric constant, and a character list element can correspond only to a character constant. If the data types of a numeric list element and its corresponding numeric constant do not match, conversion is performed according to the rules for arithmetic assignment (see Table 7-1).

A complex constant has the form of a pair of real or integer constants separated by a comma and enclosed in parentheses. Spaces can occur between the opening parenthesis and the first constant, before and after the separating comma, and between the second constant and the closing parenthesis.

A logical constant represents true or false values, that is, `.TRUE.` or any value beginning with T, `.T`, `t`, or `.t`; or `.FALSE.` or any value beginning with F, `.F`, `f`, or `.f`.

A character constant is delimited by apostrophes. An apostrophe occurring within a character constant is represented by two consecutive apostrophes.

Hollerith, octal, and hexadecimal constants are not permitted.

- A *null value*—a null value is specified by two consecutive commas with no intervening constant, or by an initial comma or trailing comma. Spaces can occur before or after the commas. A null value indicates that the corresponding list element remains unchanged. A null value can represent an entire complex constant, but cannot be used for either part of a complex constant.
- A *repetition of constants in the form r*c*—the form `r*c` indicates `r` occurrences of `c`, where `r` is a nonzero, unsigned integer constant and `c` is a constant. Spaces are not permitted except within the constant `c` as specified above.
- A *repetition of null values in the form r**—the form `r*` indicates `r` occurrences of a null value, where `r` is an unsigned integer constant.

A value separator in a record can be any one of the following:

- One or more spaces or tabs
- A comma, with or without surrounding spaces or tabs
- A slash, with or without surrounding spaces or tabs

The slash terminates processing of the input statement and the record, leaving all remaining I/O list elements unchanged.

When any of the above appear in a character constant, they are considered part of the constant, not value separators.

The end of a record is equivalent to a space character except when it occurs in a character constant. In this case, the end of the record is ignored, and the character constant is continued with the next record. That is, the last character in the previous record is followed immediately by the first character of the next record.

Spaces at the beginning of a record are ignored unless they are part of a character constant continued from the previous record. In this case, the spaces at the beginning of the record are considered part of the constant.

Each input statement reads one or more records as required to satisfy the I/O list. If a slash separator occurs or the I/O list is exhausted before all the values in a record are used, the remainder of the record is ignored.

An example of the use of list-directed READ statements follows.

A program unit consists of:

```
CHARACTER*14 C
DOUBLE PRECISION T
COMPLEX D,E
LOGICAL L,M
READ (1,*) I,R,D,E,L,M,J,K,S,T,C,A,B
  .
  .
  .
```

And the external record to be read contains:

```
4 6.3 (3.4,4.2), (3, 2) , T,F,,3*14.6 , 'ABC,DEF/GHI''JK' /
```

Upon execution of the program unit, the following values are assigned to the I/O list elements:

I/O List Element	Value
I	4
R	6.3
D	(3.4,4.2)
E	(3.0,2.0)
L	.TRUE.
M	.FALSE.
K	14
S	14.6
T	14.6D0
C	ABC,DEF/GHI JK

A, B, and J are unchanged.

11.4.1.3 Namelist-Directed Sequential READ Statement

The namelist-directed sequential READ statement performs the following operations:

- Reads data from external records accessed under the sequential mode of access until it finds the specified group-name.
- Translates the data from external to internal form using the data types of the entities in the corresponding NAMELIST statement, and the forms of the data, to provide editing.
- Assigns the translated data to the specified namelist entities in the order in which the entities appear in the input records.

An example of the namelist-directed READ statement follows.

```
NAMelist /CONTROL/ TITLE, RESET, START, STOP, INTERVAL
CHARACTER*10 TITLE
REAL*8 START, STOP
LOGICAL*4 RESET
INTEGER*4 INTERVAL
READ (UNIT=1,NML=CONTROL)
.
.
.
```

In this example, the NAMELIST statement associates the group-name CONTROL with a list of five entities. The corresponding READ statement reads input data and assigns values to specified namelist entities.

The input for a namelist-directed READ consists of a record or records delimited by the special symbol dollar sign (\$), which starts in the second column of the first record.

The namelist input has the form:

```
column 2
  ↓
$group-name entity = value [,entity = value ,...] ${END}
```

where:

\$

is the special symbol used to indicate the beginning or end of input. The ampersand (&) can be used in place of the dollar sign.

group-name

is the name of the namelist that contains the entity or entities to be given values. The namelist must have been previously defined in a NAMELIST statement in the program unit.

entity

is a namelist-defined entity. The entity can be a variable, array name, subscripted variable, variable with a substring, or subscripted variable with a substring.

value

is a constant, a list of constants, a repetition of constants in the form $r*c$, or a repetition of values in the form $r*$ (see Section 11.4.1.2).

END

is an optional part of the last delimiter.

Information on syntax rules for namelist input, prompting for current values, and assigning values is presented separately under the headings that follow.

Syntax Rules for Namelist Input

The following syntax rules apply to creating namelist input:

1. The group-name cannot contain spaces or tabs and must be contained within a single record.
2. The entities appearing on the left side of the equal sign in a value assignment cannot contain spaces or tabs except within the parentheses of a subscript or substring specifier. Each entity must be contained in a single record.
3. Each constant that appears in a value assignment has the form of the corresponding FORTRAN constant. A complex constant has the form of a pair of real or integer constants separated by a comma and enclosed in parentheses. Spaces can occur between the opening parenthesis and the first constant, before and after the separating comma, and between the second constant and the closing parenthesis.
4. A logical constant represents true or false values, that is, `.TRUE.` or any value beginning with `T`, `.T`, `t`, or `.t`; or `.FALSE.` or any value beginning with `F`, `.F`, `f`, or `.f`. A

character constant is delimited by apostrophes. An apostrophe occurring within a character constant is represented by two consecutive apostrophes. Hollerith, octal, and hexadecimal constants are not permitted.

5. The valid separators in a list of constants are spaces, tabs, and commas. Except within a character constant, any number of consecutive spaces and tabs is equivalent to a single space. A null value is specified by two consecutive commas, by an initial comma, or by a trailing comma. A separating comma preceded or followed by spaces is equivalent to a single comma. A null value indicates that the corresponding namelist array element is unchanged. A null value can represent an entire complex constant, but it cannot be used for either part of a complex constant.
6. The form $r*c$ indicates r occurrences of c , where r is a nonzero, unsigned integer constant and c is a constant. Spaces are not permitted except within the constant c in complex or character constants.
7. The form $r*$ indicates r occurrences of a null value, where r is an unsigned integer constant.
8. The valid separators in a list of value assignments are spaces, tabs, and commas. Any number of consecutive spaces and tabs is equivalent to a single space. A separating comma preceded or followed by spaces is equivalent to a single comma. Consecutive commas are not permitted.
9. The equal sign in a value assignment can be preceded and followed by any number of spaces or tabs.
10. The end of a record in namelist input is equivalent to a space character except when the end of the record occurs in a character constant. If this occurs, the end of the record is ignored, and the character constant is continued with the next record. That is, the last character in the previous record is followed immediately by the second character of the next record. The first character is used for carriage control.

Prompting for Current Values

If your program is executing a namelist READ statement, you may prompt it for the group name and namelist entities that it will accept. To do this, enter a question mark (?) record character. The group name and current values of the namelist entities for that group will then be displayed as in namelist output (see Section 11.5.1.3).

Assigning Values

Input values can be assigned in any order using an assignment of the form: entity=value. Each new line of input may begin in column 2 or in any column thereafter. Column 1 of each record is assumed to contain a FORTRAN carriage-control character, and any data placed in that column is ignored.

Assigned values, array subscripts, and substring specifiers must be constant values; use of symbolic (PARAMETER) constants is not permitted.

Input values can be any of the following data types: integer, real, logical, complex, and character. If the data type of a namelist entity and its assigned constant value do not match, conversion is performed according to the rules for arithmetic assignment (see Table 7-1). Conversion between numeric and character data types is not permitted.

An example of namelist-directed data input follows.

```
column 2
↓
#CONTROL
(TAB) TITLE='TESTT002AA',
(TAB) INTERVAL=1,
(TAB) RESET=,TRUE.,
(TAB) START=10.2,
(TAB) STOP =14.5
#END
```

Upon program execution, values are assigned to list entities with the following results:

Entity	Value
TITLE	TESTT002AA
RESET	T
START	10.2
STOP	14.5
INTERVAL	1

In this example, values were assigned to all of the namelist entities previously associated with the group-name CONTROL. However, it is not necessary to assign values to all of the list entities defined in the corresponding NAMELIST group-name.

The namelist-directed READ statement does not change the values of namelist entities that do not appear in the input data. Similarly, when character substrings and array elements are specified, only the values of the specified variable substrings and array elements are changed. For example, if the next input to the character variable TITLE used in the last example contains

```
column 2
↓
#CONTROL△TITLE(9:10)='BB'△#END
```

then its new value is TESTT002BB; the first eight positions of the variable do not change.

When a list of values is assigned to an array name, the first value in that list is assigned to the first element of the array, the second value is assigned to the second element of the array, and so on. The number of array elements assigned must be less than or equal to the size of the array. Consecutive commas within a list indicate that the values of the array elements remain unchanged. An example follows.

A program unit contains:

```
DIMENSION ARRAY(20)
NAMelist /ELEM/ ARRAY
READ (UNIT=1,NML=ELEM)
```

and the input contains:

```
column 2
↓
$ELEM
ARRAY=1.1, 1.2, , 1.4#END
```

Upon program execution, the READ statement assigns values to array elements with the following results:

Array Element	Value
ARRAY(1)	1.1
ARRAY(2)	1.2
ARRAY(3)	unchanged
ARRAY(4)	1.4
ARRAY(5) - ARRAY(20)	unchanged

When a list of values is assigned to an array element, the assignment begins with the specified array element, rather than with the first element of the array. For example, if the next input to ARRAY consists of the following:

```
column 2
↓
$ELEM
ARRAY(3)=34.54, 45.34, 87.63, 3*20.00
$END
```

Upon program execution, the READ statement assigns new values only to ARRAY elements 3 through 8; it does not alter unspecified elements.

11.4.1.4 Unformatted Sequential READ Statement

The unformatted sequential READ statement reads an external record accessed under the sequential or keyed mode of access; it assigns the fields of binary data contained in that record to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list. The data is not translated. The amount of data assigned to each element is determined by the element's data type.

The unformatted sequential READ statement reads exactly one record. If the I/O list does not use all the values in a record, the remainder of the record is discarded; this happens when there are more values in the record than elements in the list. If the number of list elements is greater than the number of values in the record, an error occurs.

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

Some examples of the use of the unformatted sequential READ statement follow.

1. In the following example, the READ statement reads one record from the file connected to logical unit 1 and assigns values of binary data to variables FIELD1 and FIELD2, in that order.

```
READ (UNIT=1) FIELD1, FIELD2
```

2. In the following example, the READ statement advances the file connected to logical unit 8 by one record.

```
READ (8)
```

11.4.2 Direct Access READ Statements

Direct access READ statements transfer input data to internal storage from external records accessed under the direct mode of access.

The formats of the two forms of direct access READ statement—*formatted* and *unformatted*—are as follows:

Formatted

```
READ (extu,rec,fmt[,iostat][,err]) [iolist]
```

Unformatted

```
READ (extu,rec[,iostat][,err]) [iolist]
```

The meanings of the symbolic abbreviations used to represent control-list parameters in the preceding command lines are as follows:

- extu—a logical unit specifier
- rec—a record specifier
- fmt—a format specifier
- iostat—an I/O status specifier
- err—transfer-of-control specifier

The I/O-list parameter is represented by the symbolic abbreviation iolist.

All of the parameters used in I/O statements are described in Sections 11.3.1 (control-list parameters) and 11.3.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 11.3.1.10.

The uses and effects of the two forms of direct access READ statements are described in Sections 11.4.2.1 and 11.4.2.2.

11.4.2.1 Formatted Direct Access READ Statement

The formatted direct access READ statement performs the following operations:

- Reads character data from one or more external records accessed under the direct mode of access.
- Translates the data from character to binary form using format specifications to provide editing.
- Assigns the translated data to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list.

If the I/O list and formatting do not use all the characters in a record, the remainder of the record is discarded; if the I/O list and formatting require more characters than are contained in the record, the remaining fields are read as spaces.

An example of the use of the formatted direct access READ statement is:

```
      READ (2,REC=35,FMT=10) (NUM(K), K=1,10)
10  FORMAT (10I2)
```

In this example, the READ and FORMAT statements read the first 10 fields from record 35 in the file connected to logical unit 2, translate the values to binary form, and then assign the translated values to the internal storage locations of the 10 elements of the array NUM.

11.4.2.2 Unformatted Direct Access READ Statement

The unformatted direct access READ statement reads an external record accessed under the direct mode of access; it assigns the fields of binary data contained in that record to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list. The data is not translated. The amount of data assigned to each element is determined by that element's data type.

The unformatted direct access READ statement reads exactly one record. If that record contains more fields than there are elements in the I/O list of the statement, the unused fields are discarded; if there are more elements than fields, an error occurs.

Examples of the use of unformatted direct access READ statements follow.

1. In the following example, the READ statement reads record 10 in the file connected to logical unit 1 and assigns binary integer values to elements 1 and 8 of the array LIST.

```
      READ (1/10) LIST(1), LIST(8)
```

2. In the following example, the READ statement reads record 58 in the file connected to logical unit 4 and assigns binary values to five elements of the array RHO.

```
      READ (4,REC=58,Iostat=k,ERR=500) (RHO(N), N=1,5)
```

11.4.3 Indexed READ Statements

The indexed READ statement transfers input data to internal storage from external records accessed under the keyed mode of access. There are two classes: *formatted* and *unformatted*.

A series of records in an indexed file can be read in key value sequence by using a sequential READ statement in conjunction with an indexed READ statement. The first record in the sequence is read using the indexed statement; the rest are read using sequential statements.

The forms of the two classes of indexed READ statement are as follows:

Formatted

```
READ (extu,fmt,key[,keyid][,iostat][,err]) [iolist]
```

Unformatted

```
READ (extu,key[,keyid][,iostat][,err]) [iolist]
```

The meanings of the symbolic abbreviations used to represent control-list parameters in the preceding command lines are as follows:

- extu—a logical unit specifier
- fmt—a format specifier
- key—a key specifier
- keyid—a key-of-reference specifier
- iostat—an I/O status specifier
- err—transfer-of-control specifier

The I/O-list parameter is represented by the symbolic abbreviation iolist.

All of the parameters used in I/O statements are described in Sections 11.3.1 (control-list parameters) and 11.3.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 11.3.1.10.

The uses and effects of the two forms of indexed READ statements are described in Sections 11.4.3.1 and 11.4.3.2.

11.4.3.1 Formatted Indexed READ Statement

The formatted indexed READ statement performs the following operations:

- Reads character data from one or more external records accessed under the keyed mode of access.
- Translates the data from character to binary form using format specifications to provide editing.
- Assigns the translated values to the elements in the I/O list, in the order, from left to right, in which they appear in the list.

The formatted indexed READ statement may be used only on indexed files. If the I/O list and format specifications specify that additional records are to be read, the statement reads those additional records sequentially using the current key-of-reference value.

If the KEYID parameter is omitted, the key-of-reference remains unchanged from the most recent specification. If the KEYID parameter is omitted from the first keyed read, the key-of-reference is the primary key.

If the specified key value is shorter than the key field referred to, the key value is matched against the leftmost characters of the appropriate key field until a match is found; the record supplying the match is then read. If the key value is longer than the key field referred to, an error occurs.

An example of the use of the formatted indexed READ statement is:

```
READ (3,KAT(25),KEY='ABCD') A,B,C,D
```

In this example the READ statement retrieves a record with a key value of 'ABCD' in the primary key, and then uses the format contained in the array item KAT(25) to read the first four fields from the record into variables A,B,C, and D.

11.4.3.2 Unformatted Indexed READ Statement

The unformatted indexed READ statement reads an external record accessed under the keyed mode of access; it assigns the fields of binary data contained in that record to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list. The data is not translated. The amount of data assigned to each element is determined by the element's data type.

The unformatted indexed READ statement reads exactly one record, and may be used only on indexed files. If the number of I/O list elements is less than the number of fields in the record being read, the unused fields in the record are discarded. If the number of I/O list elements is greater than the number of fields, an error occurs.

If a specified key value is shorter than the key field referred to, the key value is matched against the leftmost characters of the appropriate key field until a match is found; the record supplying the match is then read. If the specified key value is longer than the key field referred to, an error occurs.

Some examples of the use of the unformatted indexed READ statement follow.

```
OPEN (UNIT=3, STATUS='OLD',  
1     ACCESS='KEYED', ORGANIZATION='INDEXED',  
2     FORM='UNFORMATTED',  
3     KEY=(1:5,30:37,18:23))
```

```
READ (3,KEY='SMITH') ALPHA, BETA
```

In this example, the READ statement reads from the file connected to logical unit 3 and retrieves the record with the value 'SMITH' in the primary key field (bytes 1 to 5). The first two fields of the record retrieved are placed in variables ALPHA and BETA, respectively.

```
READ (3,KEYGE='XYZDEF',KEYID=2,ERR=99) IKEY
```

In this example, the READ statement retrieves the first record having a value equal to or greater than 'XYZDEF' in the second alternate key field (bytes 18 to 23). The first field of that record is placed in the variable IKEY.

11.4.4 Internal READ Statement

The internal READ statement transfers input data to internal storage from an internal file. The DECODE statement discussed in Appendix A may be used as an alternative to the internal READ statement. There are two classes of internal READ statement: *formatted* and *list-directed*.

The forms of the two classes of internal READ statement are as follows:

Formatted

```
READ (intu,fmt[,iostat][,err][,end]) [iolist]
```

List-Directed

```
READ (intu,*[,iostat][,err][,end]) [iolist]
```

The meanings of the symbolic abbreviations used to represent control-list parameters in the preceding command lines are as follows:

- *intu*—an internal file specifier.
- *fmt*—a format specifier.
- ***—the list-directed formatting specifier. You can also use *FMT=*.*
- *iostat*—an I/O status specifier.
- *err/end*—transfer-of-control specifiers.

The I/O-list parameter is represented by the symbolic abbreviation *iolist*.

All of the parameters used in I/O statements are described in Sections 11.3.1 (control-list parameters) and 11.3.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 11.3.1.10.

The uses and effects of the two forms of internal READ statements are described in Sections 11.4.4.1 and 11.4.4.2.

11.4.4.1 Formatted Internal READ Statement

The formatted internal READ statement performs the following operations:

- Reads character data from an internal file.
- Translates the data from character to binary form using format specifications to provide editing.
- Assigns the translated data to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list.

11.4.4.2 List-Directed Internal READ Statement

The list-directed internal READ statement performs the following operations:

- Reads character data from an internal file.
- Translates the data from external to binary form using the data types of the elements in the I/O list and the forms of the data to provide editing.
- Assigns the translated data to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list.

Namelist-directed formatting is not permitted with an internal READ statement. Refer to Section 11.2.3.2 for information on the characteristics and use of internal files.

The following program segment demonstrates the use of internal-file reads:

```
INTEGER IVAL
CHARACTER TYPE, RECORD*80
CHARACTER*(*) AFMT, IFMT, OFMT, ZFMT
PARAMETER (AFMT='(Q,A)', IFMT='(I10)', OFMT='(O11)',
1          ZFMT='(Z8)')
ACCEPT AFMT, ILEN, RECORD
TYPE = RECORD(1:1)
IF (TYPE .EQ. 'D') THEN
    READ (RECORD(2:MIN(ILEN, 11)), IFMT) IVAL
ELSE IF (TYPE .EQ. 'O') THEN
    READ (RECORD(2:MIN(ILEN, 12)), OFMT) IVAL
ELSE IF (TYPE .EQ. 'X') THEN
    READ (RECORD(2:MIN(ILEN, 9)), ZFMT) IVAL
ELSE
    PRINT *, 'ERROR'
END IF
END
```

This program segment reads a record and examines the first character to determine whether the remaining data should be interpreted as decimal, octal, or hexadecimal. It then uses internal-file reads to make appropriate conversions from character string representations to binary.

11.5 WRITE Statements

The WRITE statements transfer output data from internal storage to user-specified external logical units (disks, printers, terminals, mailboxes) or to internal files.

WRITE statements can be used in sequential, direct, keyed, or internal access modes. These forms of the WRITE statement are discussed in Sections 11.5.1 through 11.5.4.

WRITE statements cannot write to existing records in an indexed file. For statements that can perform this function in indexed files, refer to the REWRITE statement discussed in Section 11.6.

11.5.1 Sequential WRITE Statements

Sequential WRITE statements transfer output data from internal storage to external records accessed under the sequential mode of access. See Section 11.2.3.4 for descriptions of the various access modes.

The formats of the four forms of sequential WRITE statement are as follows:

Formatted

```
WRITE (extu,fmt[,iostat][,err]) [iolist]
```

List-Directed

```
WRITE (extu,*[,iostat][,err]) [iolist]
```

Namelist-Directed

```
WRITE (extu,nml[,iostat][,err])
```

Unformatted

```
WRITE (extu[,iostat][,err]) [iolist]
```

The meanings of the symbolic abbreviations used to represent control-list parameters in the preceding command lines are as follows:

- extu—a logical unit specifier.
- fmt—a format specifier.
- *—the list-directed formatting specifier. You can also use FMT=*

- nml—a namelist specifier.
- iostat—an I/O status specifier.
- err—a transfer-of-control specifier.

The I/O-list parameter is represented by the symbolic abbreviation iolist.

All of the parameters used in I/O statements are described in Sections 11.3.1 (control-list parameters) and 11.3.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 11.3.1.10.

The uses and effects of the four forms of sequential WRITE statements are described in Sections 11.5.1.1 through 11.5.1.4.

11.5.1.1 Formatted Sequential WRITE Statement

The formatted sequential WRITE statement performs the following operations:

- Retrieves specified data from internal storage.
- Translates the data from binary to character form using format specifications to provide editing.
- Writes the translated values to an external record accessed under the sequential mode of access.

The length of the records written to a user-specified output device (for example, a line printer) must not exceed the maximum record length which that device can process. In the case of a line printer, this maximum is usually 132 characters.

Using an appropriate format specification, a statement can write more than one record.

Because numeric data transferred by formatted output statements is always rounded during its conversion from binary to character form, a loss of precision may result if this data is subsequently used as input. It is recommended, therefore, that whenever numeric output is to be used subsequently as input, unformatted output and input statements be used for data transfer.

Some examples of the use of formatted sequential WRITE statements follow.

1. In the following example, the WRITE statement writes one record to logical unit 6. The record consists of the character constant defined in the FORMAT statement.

```
WRITE (6,650)
650 FORMAT (' HELLO THERE')
```

2. In the following example, the WRITE statement writes one record consisting of fields AYE, BEE, and CEE to logical unit 1.

```
WRITE (1,95) AYE, BEE, CEE
95 FORMAT (3F8.5)
```

3. In the following example, the WRITE statement writes three separate records to logical unit 1; each record consists of only one field.

```
WRITE (1,900) DEE, EEE, EFF
900 FORMAT (F8.5)
```

11.5.1.2 List-Directed Sequential WRITE Statement

The list-directed sequential WRITE statement performs the following operations:

- Retrieves specified data from internal storage.
- Translates that data from binary to character form using the data type of the elements in the I/O list to provide editing.
- Writes the translated values to an external record accessed under the sequential mode of access.

The values transferred as output by the list-directed WRITE statement have the same forms as those of the constant values transferred as input by the list-directed READ and ACCEPT statements, with the following exception: character constants are transferred without delimiting apostrophes, and each internal apostrophe is represented by only one apostrophe instead of two. As a consequence of this exception, records containing list-directed character output data can be printed, but cannot be used for list-directed input. (Refer to Section 11.4.1.2 for a full discussion of list-directed value forms.)

Table 11-5 shows the default output formats for each data type.

Table 11-5: List-Directed Output Formats

Data Type	Output Format
LOGICAL*1(BYTE)	I5
LOGICAL*2	L2
LOGICAL*4	L2
INTEGER*2	I7
INTEGER*4	I12
REAL	1PG15.7E2
REAL*8	1PG24.16E2
REAL*8(/G__FLOATING)	1PG24.15E3
REAL*16	1PG43.33E4
COMPLEX	'(',1PG14.7E2,',',1PG14.7E2,')'
COMPLEX*16	'(',1PG23.16E2,',',1PG23.16E2,')'
COMPLEX*16(/G__FLOATING)	'(',1PG23.15E3,',',1PG23.15E3,')'
CHARACTER	An (where n is the length of the character expression)

Note that:

- List-directed output statements do not produce octal values, null values, slash separators, or repeated forms of values.
- List-directed output edits a complex value so that there are no embedded spaces in the value.
- Each output record begins with a space for carriage control.
- Each output statement writes one or more complete records.
- Each individual output value is contained within a single record, with the exception of character constants longer than one record length and complex constants that can be split after the comma.

An example of the use of the list-directed WRITE statement follows.

```
DIMENSION A(4)
DATA A/4*3.4/
WRITE (1,*) 'ARRAY VALUES FOLLOW'
WRITE (1,*) A,4
```

In this example, the WRITE statements write the following records to logical unit 1:

```
ARRAY VALUES FOLLOW
  3.400000      3.400000      3.400000      3.400000      4
```

11.5.1.3 Namelist-Directed Sequential WRITE Statement

The namelist-directed sequential WRITE statement performs the following operations:

- Retrieves data specified by the namelist specifier from internal storage.
- Translates that data from internal to external form using the data type of the list entities in the corresponding NAMELIST statement.
- Writes the translated values to external records accessed under the sequential mode of access.

The namelist-directed WRITE statement transfers as output the current values of all list entities associated with the specified namelist specifier. These values are written in a form that can be read as input by the namelist-directed READ and ACCEPT statements.

The order of data output is dictated by the sequence in which namelist entities are defined in a NAMELIST statement; the first list entity and its value are written first, the second list entity and its value are written second, and so on. Each value display begins on a new line.

An example of the namelist-directed WRITE statement follows.

A program unit consists of:

```
CHARACTER*19 NAME(2)/2*' '//
REAL PITCH, ROLL, YAW, POSITION(3)
LOGICAL DIAGNOSTICS
INTEGER ITERATIONS
NAMelist /PARAM/ NAME, PITCH, ROLL, YAW, POSITION,
1      DIAGNOSTICS, ITERATIONS
.
.
.
READ (UNIT=1,NML=PARAM)
WRITE (UNIT=1,NML=PARAM)
.
.
.
```

And the input contains:

```
Δ$PARAM#NAME(2)(10:)= 'HEISENBERG',
ΔPITCH=5.0, YAW=0.0, ROLL=5.0,
ΔDIAGNOSTICS=.TRUE.,
ΔITERATIONS=10$END
```

The WRITE statement writes the following:

```
Δ$PARAM
ΔNAME      = '                ', '                ' HEISENBERG',
ΔPITCH     =      5.000000      ,
ΔROLL      =      5.000000      ,
ΔYAW       =      0.00000000E+00,
ΔPOSITION  =      3*0.00000000E+00,
ΔDIAGNOSTICS = T,
ΔITERATIONS =                10
Δ$END
```

Notice that character values are enclosed in apostrophes. The value of POSITION is not defined in the namelist-directed input. It may be defined elsewhere in the program or be undefined. The namelist-directed WRITE statement prints the current contents of POSITION.

11.5.1.4 Unformatted Sequential WRITE Statement

The unformatted sequential WRITE statement transfers specified binary data from internal storage to an external record accessed under the sequential mode of access. The data is not translated.

The unformatted sequential WRITE statement writes exactly one record; if there is no I/O list, the statement writes one null record.

Some examples of the use of the unformatted sequential WRITE statement follow.

```
WRITE (1) (LIST(K), K=1,5)
```

In this example, the WRITE statement writes a record to the file connected to logical unit 1 containing the values, in binary form, of elements 1 through 5 of the array LIST.

```
WRITE (4)
```

In this example, the WRITE statement writes one null record to the file connected to logical unit 4.

11.5.2 Direct Access WRITE Statements

Direct access WRITE statements transfer output data from internal storage to external records accessed under the direct mode of access. The OPEN statement is used to establish the attributes of a direct access file.

The formats of the two forms of direct access WRITE statement—*formatted* and *unformatted*—are as follows:

Formatted

```
WRITE (extu,rec,fmt[,iostat][,err]) [iolist]
```

Unformatted

```
WRITE (extu,rec[,iostat][,err]) [iolist]
```

The meanings of the symbolic abbreviations used to represent control-list parameters in the preceding command lines are as follows:

- extu—a logical unit specifier
- rec—a record specifier
- fmt—a format specifier
- iostat—an I/O status specifier
- err—transfer-of-control specifier

The I/O-list parameter is represented by the symbolic abbreviation iolist.

All of the parameters used in I/O statements are described in Sections 11.3.1 (control-list parameters) and 11.3.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 11.3.1.10.

The uses and effects of the two forms of direct access WRITE statements are described in Sections 11.5.2.1 and 11.5.2.2.

11.5.2.1 Formatted Direct Access WRITE Statement

The formatted direct access WRITE statement performs the following operations:

- Retrieves binary values from internal storage.
- Translates those values to character form using format specifications to provide editing.
- Writes the translated data to a user-specified external record accessed under the direct mode of access.

If the values specified by the I/O list and formatting do not fill the output record being written, the unused portion of the record is filled with space characters. If the values overflow the record, an error occurs.

11.5.2.2 Unformatted Direct Access WRITE Statement

The unformatted direct access WRITE statement retrieves binary values from internal storage and writes those values to a user-specified external record accessed under the direct mode of access. The values are not translated.

If the values specified by the I/O list do not fill the output record being written, the unused portion of the record is filled with zeros. If the values do not fit in the record, an error occurs.

11.5.3 Indexed WRITE Statements

The indexed WRITE statements transfer output data from internal storage to external records accessed under the keyed mode of access.

The indexed WRITE statement always writes a new record. The REWRITE statement (see Section 11.6) is used to update an existing record.

The OPEN statement is used to establish the attributes of an indexed file.

The syntactic form of the indexed WRITE statement is identical to that of the sequential WRITE statement; the two statements differ only in that the indexed WRITE statement refers to a logical unit connected to an indexed file, whereas the sequential WRITE statement refers to a logical unit connected to a sequential file.

The formats of the two forms of indexed WRITE statement—*formatted* and *unformatted*—are as follows:

Formatted

```
WRITE (extu,fmt[,iostat][,err]) [iolist]
```

Unformatted

```
WRITE (extu[,iostat][,err]) [iolist]
```


The meanings of the symbolic abbreviations used to represent control-list parameters in the preceding command lines are as follows:

- extu—a logical unit specifier
- fmt—a format specifier
- iostat—an I/O status specifier
- err—transfer-of-control specifier

The I/O-list parameter is represented by the symbolic abbreviation iolist.

All of the parameters used in I/O statements are described in Sections 11.3.1 (control-list parameters) and 11.3.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 11.3.1.10.

The uses and effects of the two forms of indexed WRITE statements are described in Sections 11.5.3.1 and 11.5.3.2.

11.5.3.1 Formatted Indexed WRITE Statement

The formatted indexed WRITE statement performs the following operations:

- Retrieves binary values from internal storage.
- Translates those values to character form using format specifications to provide editing.
- Writes the translated data to one or more external records accessed under the keyed mode of access.

No key parameters are required in the list of control parameters because all necessary key information is contained in the output record.

If the values specified by the I/O list and formatting do not fill a fixed-length record being written, the unused portion of the record is filled with space characters. If additional records are specified, they are inserted in the file logically according to the key values contained in each record.

An example of the use of formatted indexed WRITE statement follows.

```
WRITE (4,100) KEYVAL, (RDATA(I), I=1,20)
100 FORMAT (A10,20F15.7)
```

This example assumes that the first 10 bytes of a record are a character key. In this example, the WRITE statement writes the translated values of each of the 20 elements of the array RDATA to a new formatted record in the indexed file connected to logical unit 4, with KEYVAL being the key by which the record is accessed.

When you write an INTEGER key using the formatted indexed WRITE statement, the key is translated from internal binary form to external character form. A subsequent attempt to read the record using an integer key may not match the key field in the record.

11.5.3.2 Unformatted Indexed WRITE Statement

The unformatted indexed WRITE statement retrieves binary values from internal storage and writes those values to an external record accessed under the keyed mode of access. The values are not translated.

No key parameters are required in the list of control parameters because all necessary key information is contained in the output record.

If the values specified by the I/O list do not fill a fixed-length record being written, the unused portion of the record is filled with zeros; if the values specified overflow the record, an error occurs. The use of records, or structured data items, (see Section 6.2.5) is advantageous when writing to indexed files. Such files usually have a fixed record format. By using a structure declaration that models the file record format, I/O can be accomplished with a single record variable—instead of a potentially long I/O list. For an example, see Section 14.3.

11.5.4 Internal WRITE Statement

The internal WRITE statement transfers output data from internal storage to an internal file. You can also use the ENCODE statement discussed in Appendix A to control internal output.

Refer to Section 11.2.3.2 for information on the characteristics and use of internal files.

The formats of the two forms of internal WRITE statements—*formatted* and *list-directed*—are as follows:

Formatted

```
WRITE (intu,fmt[,iostat][,err]) [iolist]
```

List-Directed

```
WRITE (intu,*[,iostat][,err]) [iolist]
```

The meanings of the symbolic abbreviations used to represent control-list parameters in the preceding command lines are as follows:

- *intu*—an internal file specifier.
- *fmt*—a format specifier.
- ***—the list-directed formatting specifier. You can also use `FMT=*`.
- *iostat*—an I/O status specifier.
- *err*—transfer-of-control specifier.

The I/O-list parameter is represented by the symbolic abbreviation *iolist*.

All of the parameters used in I/O statements are described in Sections 11.3.1 (control-list parameters) and 11.3.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 11.3.1.10.

Namelist-directed formatting is not permitted with internal WRITE statements.

The uses and effects of the two forms of internal WRITE statements are described in Sections 11.5.4.1 and 11.5.4.2.

11.5.4.1 Formatted Internal WRITE Statement

The formatted internal WRITE statement performs the following operations:

- Retrieves data from internal storage.
- Translates that data from binary to character form using format specifications to provide editing.
- Writes the translated values to an internal file.

11.5.4.2 List-Directed Internal WRITE Statement

The list-directed internal WRITE statement performs the following operations:

- Retrieves data from internal storage.
- Translates that data from binary to character form using the data type of the elements in the I/O list to provide editing.
- Writes the translated values to an internal file.

11.6 REWRITE Statement

The REWRITE statement transfers output data from internal storage to the current record in a file with indexed or relative organization.

The REWRITE statement transfers output data from internal storage to a specified record in a file, with indexed or relative organization, which was most recently accessed by a READ statement. The OPEN statement establishes the attributes of the file.

The formats of the two forms of REWRITE statement—*formatted* and *unformatted*—are as follows:

Formatted

```
REWRITE (extu,fmt[,iostat][,err]) [iolist]
```

Unformatted

```
REWRITE (extu[,iostat][,err]) [iolist]
```

The meanings of the symbolic abbreviations used to represent control-list parameters in the preceding command lines are as follows:

- extu—a logical unit specifier
- fmt—a format specifier
- iostat—an I/O status specifier
- err—transfer-of-control specifier

The I/O-list parameter is represented by the symbolic abbreviation iolist.

All of the parameters used in I/O statements are described in Sections 11.3.1 (control-list parameters) and 11.3.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 11.3.10.

The uses and effects of the two forms of REWRITE statements are described in Sections 11.6.1 and 11.6.2.

11.6.1 Formatted REWRITE Statement

The formatted REWRITE statement performs the following operations:

- Retrieves binary values from internal storage.
- Translates those values to character form using format specifiers to provide editing.
- Writes the translated data to an existing record in a file with indexed or relative organization.

The record written to is the current record in the file, that is, the last record to be accessed by a preceding indexed or sequential READ statement. Note that changing the primary key value usually results in an error, and that attempting to rewrite more than one record in a single REWRITE statement operation causes an error. Any unused space in a rewritten fixed-length record is filled with spaces; if the record is too long, an error occurs.

An example of the use of a formatted REWRITE statement follows.

```
        REWRITE (3,10,ERR=99) NAME, AGE, BIRTH
10      FORMAT (A16,I2,AB)
```

In this example, the REWRITE statement updates the current record contained in the indexed organization file connected to logical unit 3 with the values represented by NAME, AGE, and BIRTH.

11.6.2 Unformatted REWRITE Statement

The unformatted REWRITE statement retrieves binary values from internal storage and writes those values to an existing record in a file with indexed or relative organization. The values are not translated.

The record written to is the current record in the file, that is, the last record to be accessed by a preceding indexed or sequential READ statement. Note that changing the primary key value usually results in an error. Any unused space in a rewritten fixed-length record is filled with zeros; if the record is too long, an error occurs.

11.7 ACCEPT Statement

The ACCEPT statement transfers input data to internal storage from external records accessed under the sequential mode of access. ACCEPT statements can only be used on implicitly connected logical units.

The formats of the ACCEPT statement are as follows:

```
ACCEPT f[,iolist]
```

```
ACCEPT *[,iolist]
```

```
ACCEPT n
```

The meanings of the symbolic abbreviations used to represent control-list parameters in the preceding command lines are as follows:

- f—the nonkeyword form of a format specifier
- *—the list-directed formatting specifier
- n—the nonkeyword form of a namelist specifier

The I/O-list parameter is represented by the symbolic abbreviation iolist.

All of the parameters used in I/O statements are described in Sections 11.3.1 (control-list parameters) and 11.3.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 11.3.1.10.

The ACCEPT statement functions exactly as the sequential READ statements discussed in Sections 11.4.1.1 through 11.4.1.3, with the following important exception: the ACCEPT statement can never be connected to user-specified logical units.

An example of the use of the formatted ACCEPT statement follows.

```
CHARACTER*10 CHARAR(5)
ACCEPT 200, CHARAR
200 FORMAT (5A10)
```

In this example, the ACCEPT statement reads character data from the implicit unit and assigns binary values to each of the five elements of the array CHARAR.

11.8 TYPE and PRINT Statements

The TYPE and PRINT statements transfer output data from internal storage to external records accessed under the sequential mode of access.

The formats of the TYPE and PRINT statements are as follows:

```
TYPE f[,iolist]
PRINT f[,iolist]
```

```
TYPE *[,iolist]
PRINT *[,iolist]
```

```
TYPE n
PRINT n
```

The meanings of the symbolic abbreviations used to represent control-list parameters in the preceding command lines are as follows:

- f—the nonkeyword form of a format specifier
- *—the list-directed formatting specifier
- n—the nonkeyword form of a namelist specifier

The I/O-list parameter is represented by the symbolic abbreviation iolist.

All of the parameters used in I/O statements are described in Sections 11.3.1 (control-list parameters) and 11.3.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 11.3.1.10.

TYPE and PRINT statements function exactly as the formatted sequential WRITE statement discussed in Section 11.5.1.1, with the following important exception: The formatted sequential TYPE and PRINT statements can never be used to transfer data to user-specified logical units.

An example of the use of a formatted sequential PRINT statement follows.

```
CHARACTER*16 NAME, JOB
PRINT 400, NAME, JOB
400 FORMAT ('NAME=',A,'JOB=',A)
```

In this example, the PRINT statement writes one record to the implicit output device; the record consists of four fields of character data.

Chapter 12

FORMAT Statements

A **FORMAT** statement specifies the format in which data is to be transferred as well as the data conversion (editing) required to achieve that format. **FORMAT** statements are non-executable statements used with formatted I/O statements, **ASSIGN** statements, and with **ENCODE** and **DECODE** statements.

Information related to the **FORMAT** statement is organized as follows in this chapter:

- The syntax of **FORMAT** statements (Section 12.1)
- The **FORMAT** statement's field and edit descriptors (Section 12.2)
- The carriage control options for output records (Section 12.3)
- The functions of the field separators (comma and slash) (Sections 12.4 and 12.5)
- The use of a run-time format instead of a **FORMAT** statement to create a format dynamically during program execution (Section 12.6)
- The interaction between the format specifier and the I/O list (Section 12.7)
- A summary of the rules for writing **FORMAT** statements (Section 12.8)

12.1 Syntax of **FORMAT** Statement

FORMAT statements have the form:

```
FORMAT (q1f1s1f2s2 ... fnqn)
```

where:

q

is zero or more slash (/) record terminators.

f

is a field descriptor or a group of field descriptors enclosed in parentheses.

s

is a field separator.

The entire list of field descriptors and field separators, including the parentheses, is called the format specification.

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

[r]c [r]cw [r]cw.m [r]cw.d[Ee]

where:

r

is the repeat count for the field descriptor. If you omit r, the repeat count is assumed to be 1.

c

is a format code (I,O,Z,F,E,D,G,L,A,H,X,T,P,Q,\$,.,BN,BZ,S,SP,SS, TL, or TR).

w

is the external field width, in characters.

m

is the minimum number of characters that must appear within the field (including leading zeros).

d

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

E

in this context, identifies an exponent field.

e

is the number of characters in the exponent.

The terms r, w, m, and d must all be unsigned integer constants or variable format expressions. The values of r and w must be greater than zero and less than or equal to 32767, and the values of m, d, and e must be greater than zero and less than or equal to 255. The r term is optional; however, you cannot use it in some field descriptors (see Section 12.2.1). The d and e terms are required in some field descriptors and are invalid in others. You are not allowed to use PARAMETER constants for the terms r, w, m, d, or e.

The field descriptors are:

- Integer—Iw, Ow, Zw, Iw.m, Ow.m, Zw.m
- Logical—Lw
- Real and complex—Fw.d, Ew.d, Dw.d, Gw.d, Ew.dEe, Gw.dEe
- Character—Aw
- Editing, and character and Hollerith constants—nH, '...', nX, Tn, TLn, TRn, nP, Q, \$, :, BN, BZ, S, SP, SS (n is the number of characters or character positions).

12.2 Field and Edit Descriptors

A field descriptor describes the size and format of a data item or of several data items; each data item in the external medium is called an external field. An edit descriptor specifies an editing function to be performed on a data item or items.

The numeric field descriptors ignore leading spaces in the external field. Embedded and trailing spaces are ignored only if the BN edit descriptor is specified or if BLANK='NULL' is in effect for the logical unit. Otherwise, embedded and trailing spaces are treated as zeros.

At the beginning of the execution of each formatted input statement, the BLANK attribute for the relevant logical unit determines the interpretation of spaces; the VAX FORTRAN defaults are BLANK='NULL' when an OPEN has been done and BLANK='ZERO' when no explicit OPEN has been done. During the execution of a formatted input statement, the BN and BZ edit descriptors may supersede the default interpretation of blanks. The BN and BZ edit descriptors affect only the formatted I/O statement of which they are a part (as do the S, SP, and SS edit descriptors).

Sections 12.2.1 through 12.2.12 describe each of the field and edit descriptors in detail.

12.2.1 Repeat Counts and Group Repeat Counts

You can apply the field descriptors I, O, Z, F, E, D, G, L, and A to a number of successive data fields by preceding the field descriptor with an unsigned integer constant (parameter constants not allowed) specifying the number of repetitions. This constant is called a repeat count. For example, the following two statements are equivalent:

```
20  FORMAT (E12.4,E12.4,E12.4,I5,I5,I5,I5)
```

```
20  FORMAT (3E12.4,4I5)
```

Similarly, you can apply a group of field descriptors repeatedly to data fields by enclosing these field descriptors in parentheses and preceding them with an unsigned integer constant (parameter constants not allowed). The integer constant is called a group repeat count. For example, the following two statements are equivalent:

```
50  FORMAT (2I8,3(F8.3,E15.7),2(I5))
```

```
50  FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7,I5,I5)
```

1 2 3

An H or Q field descriptor, which could not otherwise be repeated, can be enclosed in parentheses and treated as a group repeat specification. Thus, it could be repeated a desired number of times.

If you do not specify a group repeat count, a default count of 1 is assumed.

Section 12.7 discusses how to use parentheses when the number of values to be formatted is greater than the number of format specifications.

12.2.2 Variable Format Expressions

By enclosing an arithmetic expression in angle brackets, you can use it in a FORMAT statement wherever you can use an integer (except as the specification of the number of characters in the H field). For example:

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

When the format is scanned, the preceding statement performs an I (integer) data transfer with a field width of J+1. The expression is reevaluated each time it is encountered in the normal format scan.

The following rules govern the use of variable format expressions:

- If the expression is not of integer data type, it is converted to integer data type before being used.
- The expression can be any valid FORTRAN expression, including function calls and references to dummy arguments.
- The value of a variable format expression must obey the restrictions on magnitude applying to its use in the format, or an error occurs.
- Variable format expressions are not permitted in run-time formats.

Variable format expressions are evaluated each time they are encountered in the scan of the format. If the value of the variable used in the expression changes during the execution of the I/O statement, the new value is used the next time the format item containing the expression is processed. See Section 12.7 for a description of the synchronization of I/O lists with formats.

Figure 12-1 shows an example of a variable format expression.

```
DIMENSION A(5)
DATA A/1.,2.,3.,4.,5./

DO 10 I=1,10
WRITE (6,100) I
100 FORMAT (I<MAX(I,5)>)
10 CONTINUE

DO 20 I=1,5
WRITE (6,101) (A(I), J=1,I)
101 FORMAT (<I>F10.<I-1>)
20 CONTINUE
END
```

On execution, these statements produce the following output:

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

Figure 12-1: Variable Format Expression Example

12.2.3 Blank Control Editing

The treatment of embedded and trailing blanks within numeric input files is controlled by BN and BZ edit descriptors.

12.2.3.1 BN Edit Descriptor

The BN descriptor causes the processor to ignore all the embedded and trailing blanks it encounters within a numeric input field. It has the form:

BN

The effect is that of actually removing the blanks and right-justifying the remainder of the field. A field of all blanks is treated as zero. The BN descriptor affects only I, O, Z, F, E, D, and G editing during the execution of an input statement.

12.2.3.2 BZ Edit Descriptor

The BZ descriptor causes the processor to treat all the embedded and trailing blanks it encounters within a numeric input field as zeros. It has the form:

BZ

The BZ descriptor affects only I, O, Z, F, E, D, and G editing during the execution of an input statement.

12.2.4 Sign Control Editing

The treatment of optional plus characters in output data is controlled by SP, SS, and S edit descriptors.

12.2.4.1 SP Edit Descriptor

An SP descriptor causes the processor to produce a plus character (+) in any position where this character would otherwise be optional. It has the form:

SP

The SP descriptor affects only I, F, E, D, and G editing during the execution of an output statement.

12.2.4.2 SS Edit Descriptor

The SS descriptor causes the processor to suppress a leading plus character from any position where this character would normally be produced as an optional character; it has the opposite effect of the SP field descriptor described above. The SS descriptor has the form:

SS

The SS descriptor affects only I, F, E, D, and G editing during the execution of an output statement.

12.2.4.3 S Edit Descriptor

The S edit descriptor reinvokes optional plus characters (+) in numeric output fields. It has the form:

S

The S descriptor counters the action of either the SP or SS descriptor by restoring to the processor the discretion of producing plus characters on an optional basis.

The same restrictions apply as for the SP and SS descriptors.

12.2.5 Integer Editing

Integer editing is controlled by I (decimal), O (octal), and Z (hexadecimal) field descriptors.

12.2.5.1 I Field Descriptor

The I field descriptor transfers decimal integer values. It has the form:

Iw[.m]

The corresponding I/O list element must be of either integer or logical data type.

Input Processing

In an input statement, the I field descriptor transfers *w* characters from the external field and assigns them to the corresponding I/O list element as an integer value. The external data must have the form of an integer constant; it cannot contain a decimal point or exponent field.

If the value of the external field exceeds the range of the corresponding list element, an error occurs. If the first nonblank character of the external field is a minus sign, the field is treated as a negative value. If the first nonblank character is a plus sign, or if no sign appears in the field, the field is treated as a positive value. An all-blank field is treated as a value of zero.

Input Example:

Format	External Field	Internal Value
I4	2788	2788
I3	-26	-26
I9	△△△△△312	312

Output Processing

In an output statement, the I field descriptor transfers the value of the corresponding I/O list element, right-justified, to an external field that is *w* characters long. If the value does not fill the field, leading spaces are inserted; if the value is too large for the field, the entire field is filled with asterisks. If the value of the list element is negative, the field will have a minus sign as its leftmost, nonblank character. The term *w* must therefore be large enough to provide for a minus sign, when necessary. If *m* is present, the external field consists of at least *m* digits, and is zero-filled on the left, if necessary.

Input Example:

Format	Internal Value	External Representation
I3	284	284
I4	-284	-284
I5	174	△△174
I2	3244	**
I3	-473	***
I7	29.812	Not permitted: error
I4.2	1	△△01
I4.4	1	0001

Note that if *m* is zero, and the internal representation is zero, the external field is blank-filled.

12.2.5.2 O Field Descriptor

The O field descriptor transfers octal (base 8) values and can be used with any data type. It has the form:

Ow[m]

Input Processing

In an input statement, the O field descriptor transfers w characters from the external field and assigns them as an octal value to the corresponding I/O list element. The external field can contain only the numerals 0 through 7; it cannot contain a sign, a decimal point, or an exponent field. An all-blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

Input Example:

Format	External Field	Internal Octal Value
O5	32767	32767
O4	16234	1623
O3	97△	Not permitted: error

Output Processing

In an output statement, the O field descriptor transfers the octal value of the corresponding I/O list element, right-justified, to an external field that is w characters long. No signs are transmitted; a negative value is transmitted in internal form. If the value does not fill the field, leading spaces are inserted; if the value is too large for the field, the entire field is filled with asterisks. If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary.

Output Example:

Format	Internal (Decimal) Value	External Representation
O6	32767	△77777
O6	-32767	100001
O2	14261	**
O4	27	△△33
O5	10.5	41050
O4.2	7	△△07
O4.4	7	0007

Note that if m is zero, and the external representation is zero, the external field is blank-filled.

12.2.5.3 Z Field Descriptor

The Z field descriptor transfers hexadecimal (base 16) values, and can be used with any data type. It has the form:

Zw[.m]

Input Processing

In an input statement, the Z field descriptor transfers w characters from the external field and assigns them as a hexadecimal value to the corresponding I/O list element. The external field can contain only the numerals 0 through 9 and the letters A (a) through F (f); it cannot contain a sign, a decimal point, or an exponent field. An all-blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

Input Example:

Format	External Field	Internal Hexadecimal Value
Z3	A94	A94
Z5	A23DEF	A23DE
Z5	95.AF2	Not permitted: error

Output Processing

In an output statement, the Z field descriptor transfers the hexadecimal value of the corresponding I/O list element, right-justified, to an external field that is w characters long. No signs are transmitted; a negative value is transmitted in internal form. If the value does not fill the field, leading spaces are inserted; if the value is too large for the field, the entire field is filled with asterisks. If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary.

Output Example:

Format	Internal (Decimal) Value	External Representation
Z4	32767	7FFF
Z5	-32767	△8001
Z2	16	10
Z4	-10.5	C228
Z3.3	2708	A94
Z6.4	2708	△△0A94

Note that if m is zero, and the internal representation is zero, the external field is blank-filled.

12.2.6 Real Editing

Editing performed on data with a real data type is controlled by F, E, D, and G field descriptors.

NOTE

When attempting to parse textual input, you should not mix in the use of F, E, D, or G format descriptors. These descriptors accept some forms that are purely textual as valid numeric input values. For example, the input values D, E, E1, plus sign (+), minus sign (-), and period (.) are all treated as 0.0.

12.2.6.1 F Field Descriptor

The F field descriptor transfers real values. It has the form:

Fw.d

The corresponding I/O list element must be of real data type, or it must be either the real or the imaginary part of a complex data type.

Input Processing

In an input statement, the F field descriptor transfers w characters from the external field and assigns them, as a real value, to the corresponding I/O list element. If the first non-blank character of the external field is a minus sign, the field is treated as a negative value. If the first nonblank character is a plus sign, or if no sign appears in the field, the field is treated as a positive value. An all-blank field is treated as a value of zero. A field with only an exponent or decimal point is treated as a value of zero.

If the field contains neither a decimal point nor an exponent, it is treated as a real number of w digits, in which the rightmost d digits are to the right of the decimal point, with leading zeros assumed if necessary. If the field contains an explicit decimal point, the location of that decimal point overrides the location specified by the field descriptor. If the field contains an exponent, that exponent is used to establish the magnitude of the value before it is assigned to the list element.

Input Example:

Format	External Field	Internal Value
F8.5	123456789	123.45678
F8.5	-1234.567	-1234.56
F8.5	2477E+2	2477.0
F5.2	123.45	123.45

Output Processing

In an output statement, the F field descriptor transfers the value of the corresponding I/O list element, rounded to d decimal positions and right-justified, to an external field that is w characters long. If the value does not fill the field, leading spaces are inserted; if the value is too large for the field, the entire field is filled with asterisks.

The term w must be large enough to include all of the following: a minus sign when necessary (plus signs are optional); at least one digit to the left of the decimal point; the decimal point; and d digits to the right of the decimal. Therefore, w must be greater than or equal to d+3.

Output Example:

Format	Internal Value	External Representation
F8.5	2.3547188	△2.35472
F9.3	8789.7361	△8789.736
F2.1	51.44	**
F10.4	-23.24352	△△-23.2435
F5.2	325.013	*****
F5.2	-.2	-0.20

12.2.6.2 E Field Descriptor

The E field descriptor transfers real values in exponential form. It has the form:

$$Ew.d[Ee]$$

The corresponding I/O list element must be of real data type, or it must be either the real or the imaginary part of a complex data type.

Input Processing

In an input statement, the E field descriptor transfers w characters from the external field and assigns them as a real value to the corresponding I/O list element. The F field descriptor interprets and assigns data in exactly the same way.

Input Example:

Format	External Field	Internal Value
E9.3	734.432E3	734432.0
E12.4	△△1022.43E-6	1022.43E-6
E15.3	52.3759663△△△△△	52.3759663
E12.5	210.52710+10	210.5271E10

Note that in the last example, the E field descriptor treats the D exponent field indicator as an E indicator if the I/O list element is single precision.

Output Processing

In an output statement, the E field descriptor transfers the value of the corresponding I/O list element, rounded to d decimal digits and right-justified, to an external field that is w characters long. If the value does not fill the field, leading spaces are inserted; if the value is too large for the field, the entire field is filled with asterisks.

When you use the E field descriptor, data output is transferred in a standard form. This form consists of the following: a minus sign when necessary (plus signs are optional); a zero; a decimal point; d digits to the right of the decimal point; and an e+2-character exponent. The exponent has one of the following forms:

$$\left\{ \begin{array}{l} E+nn \\ E-nn \end{array} \right\} \quad Ew.d \text{ (for: exponent .LE. 99)}$$

$$\left\{ \begin{array}{l} +nnn \\ -nnn \end{array} \right\} \quad Ew.d \text{ (for: 99 .GT. exponent .LE. 999)}$$

$$\left\{ \begin{array}{l} E+n_1n_2\dots n_e \\ E-n_1n_2\dots n_e \end{array} \right\} \quad Ew.dEe$$

The exponent field width specification is optional; if it is omitted, the value of e defaults to two. If the exponent value is too large to be converted into one of the preceding forms, an error occurs.

The d digits to the right of the decimal point represent the entire value, scaled to a decimal fraction.

The term w must be large enough to include all of the following: a minus sign when necessary (plus signs are optional); a zero; a decimal point, d digits; and an exponent. Therefore, w must be greater than or equal to d+7, or to d+e+5 if e is present.

Output Example:

Format	Internal Value	External Representation
E9.2	475867.222	△0.48F+06
E12.5	475867.222	△0.47587E+06
E12.3	0.00069	△△△0.690E-03
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****
E14.5E4	-1.001	-0.10010E+0001
E14.3E6	0.000123	△0.123E-000003

12.2.6.3 D Field Descriptor

The D field descriptor transfers real values in exponential form. It has the form:

Dw.d

The corresponding I/O list element must be of real data type, or it must be either the real or the imaginary part of a complex data type.

Input Processing

In an input statement, the D field descriptor transfers w characters from the external field and assigns them as a real value to the corresponding I/O list element. The F and E field descriptors interpret and assign data in exactly the same way.

Input Example:

Format	External Field	Internal Value
BZ,D10.2	12345△△△△△	12345000.0D0
D10.2	△△123.45△△	123.45D0
D15.3	367.4981763D+04	3.674981763D+06

Output Processing

In an output statement, the D field descriptor has the same effect as the E field descriptor, except that the D exponent field indicator is used in place of the E indicator.

Output Example:

Format	Internal Value	External Representation
D14.3	0.0363	△△△△△0.363D01
D23.12	5413.87625793	△△△△△0.541387625793D+04
D9.6	1.2	*****

12.2.6.4 G Field Descriptor

The G field descriptor transfers real values in a form that, in effect, combines the F and E field descriptors. It has the form:

Gw.d[Ee]

The corresponding I/O list element must be of real data type, or it must be either the real or the imaginary part of a complex data type.

Input Processing

In an input statement, the G field descriptor transfers w characters from the external field and assigns them as a real value to the corresponding I/O list element. The F, D, and E field descriptors interpret and assign data in exactly the same way.

Output Processing

In an output statement, the G field descriptor transfers the value of the corresponding I/O list element, rounded to d decimal positions and right-justified, to an external field that is w characters long. The form in which the value is written is a function of the magnitude of the value, as described in Table 12-1.

Table 12-1: Effect of Data Magnitude on G Format Conversions

Data Magnitude	Effective Conversion
m .LT. 0.1	Ew.d[Ee]
0.1 .LE. m .LT. 1.0	F(w-4).d, n(' ')
1.0 .LE. m .LT. 10.0	F(w-4).(d-1), n(' ')
.	.
.	.
.	.
10**d-2 .LE. m .LT. 10**d-1	F(w-4).1, n(' ')
10**d-1 .LE. m .LT. 10**d	F(w-4).0, n(' ')
m .GE. 10**d	Ew.d[Ee]

The n(' ') field descriptor, which is, in effect, inserted by the G field descriptor for values within its range, specifies that four or e+2 spaces are to follow the numeric data representation.

The term w must be large enough to include all of the following: a minus sign when necessary (plus signs are optional); a decimal point; one digit to the left of the decimal point; d digits to the right of the decimal point; and either a 4-character or e+2-character exponent. Therefore, w must be greater than or equal to 1+d+7 or 1+d+5+e.

Output Example:

Format	Internal Value	External Representation
G13.6	0.01234567	Δ0,123457E-01
G13.6	-0.12345678	-0,123457ΔΔΔΔ
G13.6	1.23456789	ΔΔ1,23456ΔΔΔΔ
G13.6	12.34567890	ΔΔ12,3457ΔΔΔΔ
G13.6	123.45678901	ΔΔ123,457ΔΔΔΔ
G13.6	-1234.56789012	Δ-1234,57ΔΔΔΔ
G13.6	12345.67890123	ΔΔ12345,7ΔΔΔΔ
G13.6	123456.78901234	ΔΔ123457,ΔΔΔΔ
G13.6	-1234567.89012345	-0,123457E+07

Compare the above example with the following example, which shows the same values output using an equivalent F field descriptor.

Format	Internal Value	External Representation
F13.6	0.01234567	ΔΔΔΔΔ0,012346
F13.6	-0.12345678	ΔΔΔΔ-0,123457
F13.6	1.23456789	ΔΔΔΔΔ1,234568
F13.6	12.34567890	ΔΔΔΔΔ12,345679
F13.6	123.45678901	ΔΔΔΔ123,456789
F13.6	-1234.56789012	Δ-1234,567890
F13.6	12345.67890123	Δ12345,678901
F13.6	123456.78901234	123456,789012
F13.6	-1234567.89012345	*****

12.2.6.5 Complex Data Editing

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

Input Processing

In an input statement, the two successive fields are read and assigned to a complex I/O list element as its real and imaginary parts, respectively.

Input Example:

Format	External Field	Internal Value
F8.5,F8.5	1234567812345,67	123.45678, 12345.67
E9.1,F9.3	734,432E8123456789	734.432E8, 123456.789

Output Processing

In an output statement, the two parts of a complex value are transferred under the control of repeated or successive field descriptors. The two parts are transferred consecutively, without punctuation or spacing, unless the format specifier states otherwise.

Output Example:

Format	Internal Value	External Representation
2F8.5	2.3547188, 3.456732	$\Delta 2.35472 \Delta 3.45673$
E9.2, ' Δ, Δ ', E5.3	47587.222, 56.123	$\Delta 0.48E+06 \Delta. \Delta*****$

12.2.7 Scale Factor Editing—P Edit Descriptor

The scale factor lets you alter, during input or output, the location of the decimal point in real values and in the two parts of complex values.

The scale factor has the form:

nP

where:

n

is a signed or unsigned integer constant in the range -128 through 127. It specifies the number of positions, to the left or right, that the decimal point is to move.

A scale factor can appear anywhere in a format specification, but must precede the first field descriptor that is to be associated with it. For example:

nPFw.d nPEw.d nPDw.d nPGw.d

Input Processing

On input, the scale factor in any of the above field descriptors multiplies the data by 10^{*-n} and assigns it to the corresponding I/O list element. For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left. A -2P scale factor multiplies an input value by 100, moving the decimal point two places to the right. However, if the external field contains an explicit exponent, the scale factor has no effect.

Input Example:

Format	External Field	Internal Value
3PE10.5	$\Delta\Delta\Delta 37.614\Delta$.037614
3PE10.5	$\Delta\Delta 37.614E2$	3761.4
-3PE10.5	$\Delta\Delta\Delta\Delta 37.614$	37614.0

Output Processing

On output, the effect of the scale factor depends on the type of field descriptor associated with it. For the F field descriptor, the value of the I/O list element is multiplied by 10^{**n} before transfer to the external record. Thus, a positive scale factor moves the decimal point to the right; a negative scale factor moves the decimal point to the left.

For the E or D field descriptor, the basic real constant part of the I/O list element is multiplied by 10^{**n} , and n is subtracted from the exponent. For a positive scale factor, n must be less than $(d + 2)$ or an output conversion error occurs. Thus, a positive scale factor moves the decimal point to the right and decreases the exponent; a negative scale factor moves the decimal point to the left and increases the exponent.

Output Example:

Format	Internal Value	External Representation
1PE12.3	-270.139	△△-2.701E+02
1PE12.2	-270.139	△△△-2.70E+02
-1PE12.2	-270.139	△△△-0.03E+04

The effect of the scale factor for the G field descriptor is suspended if the magnitude of the data to be output is within the effective range of the descriptor, because the G field descriptor supplies its own scaling function. The G field descriptor functions as an E field descriptor if the magnitude of the data value is outside its range. In this case, the scale factor has the same effect as for the E field descriptor.

On input, and on output under F field descriptor control, a scale factor actually alters the magnitude of the data. On output, a scale factor under E, D, or G field descriptor control merely alters the form in which the data is transferred. In addition, on input, a positive scale factor moves the decimal point to the left, and a negative scale factor moves the decimal point to the right; on output, the effect is the reverse.

If you do not specify a scale factor with a field descriptor, a default scale factor of zero is assumed. Once you specify a scale factor, however, it applies to all subsequent real field descriptors in the same FORMAT statement, unless another scale factor appears. For example:

```
        DIMENSION A(G)
        DO 10 I=1,6
10      A(I) = 25.
        TYPE 100,A
100    FORMAT(' ',F8.2,2PF8.2,F8.2)
```

produces the following:

```
    25.00 2500.00 2500.00
2500.00 2500.00 2500.00
```

If a second scale factor appears in the FORMAT statement, it takes control from the first scale factor.

Format reversion has no effect on the scale factor (see Section 12.7). A scale factor of zero can be reinstated only by an explicit OP specification.

12.2.8 Logical Editing—L Edit Descriptor

The L field descriptor transfers logical data. It has the form:

Lw

The corresponding I/O list element must be of either integer or logical data type.

Input Processing

In an input statement, the L field descriptor transfers w characters from the external field. If the first nonblank characters of the field are T, t, .T, or .t, the value .TRUE. is assigned to the corresponding I/O list element; if the first nonblank characters are F, f, .F, or .f, the value .FALSE. is assigned. An all-blank field is assigned the value .FALSE. Any other value in the external field produces an error. Note that the logical constants .TRUE. and .FALSE. are acceptable input forms.

Output Processing

In an output statement, the L field descriptor transfers either the letter T (if the value of the corresponding I/O list element is .TRUE.) or the letter F (if the value is .FALSE.) to an external field that is w characters long. The letter T or F is in the rightmost position of the field, preceded by w-1 spaces.

Output Example:

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

12.2.9 Character Editing

Editing data with a character data type is controlled by the A and H field descriptors.

12.2.9.1 A Field Descriptor

The A field descriptor transfers character or Hollerith values. It has the form:

A[w]

The corresponding I/O list element can be of any data type. If it is of character data type, character data is transmitted. If it is of any other data type, Hollerith data is transmitted.

The value of w must be less than or equal to 32767.

Input Processing

In an input statement, the A field descriptor transfers w characters from the external record and assigns them to the corresponding I/O list element. The maximum number of characters that can be stored depends on the size of the I/O list element. For character I/O list elements, the size is the length of either the character variable, the character substring reference, or the character array element. For numeric I/O list elements, the size depends on the data type, as follows:

I/O List Element	Maximum Number of Characters
BYTE	1
LOGICAL*1	1
LOGICAL*2	2
LOGICAL*4	4
INTEGER*2	2
INTEGER*4	4
REAL	4
REAL*8(DOUBLE PRECISION)	8
REAL*16	16
COMPLEX	8 ¹
COMPLEX*16(DOUBLE COMPLEX)	16 ¹

¹ Because complex values are treated as pairs of real numbers, complex data editing requires two format codes. See Section 12.2.6.5.

If w is greater than the maximum number of characters that can be stored in the corresponding I/O list element, only the rightmost characters are assigned to that element. The leftmost excess characters are ignored. If w is less than the number of characters that can be stored, w characters are assigned to the list element, left-justified, and trailing spaces are added to fill the element.

Input Example:

Format	External Field	Internal Representation
A6	PAGE△#	# (CHARACTER*1)
A6	PAGE△#	E△# (CHARACTER*3)
A6	PAGE△#	PAGE△# (CHARACTER*6)
A6	PAGE△#	PAGE△#△△ (CHARACTER*8)
A6	PAGE△#	# (LOGICAL*1)
A6	PAGE△#	△# (INTEGER*2)
A6	PAGE△#	GE△# (REAL)
A6	PAGE△#	PAGE△#△△ (REAL*8)

Output Processing

In an output statement, the A field descriptor transfers the contents of the corresponding I/O list element to an external field w characters long. If w is greater than the list element size, the data appears in the field, right-justified, with leading spaces. If w is less than the list element, only the leftmost w characters are transferred.

Output Example:

Format	Internal Value	External Representation
A5	OHMS	△OHMS
A5	VOLTS	∇VOLTS
A5	AMPERES	AMPER

If you omit w in an A field descriptor, a default value is supplied. If the I/O list element is of character data type, the default value is the length of the I/O list element. If the I/O list element is of numeric data type, the default value is the maximum number of characters that can be stored in a variable of that data type.

12.2.9.2 H Field Descriptor

The H field descriptor transfers data between the external record and the H field descriptor itself. It has the form of a Hollerith constant:

$$nHc_1c_2c_3 \dots c_n$$

where:

n

is the number of characters to be transferred.

c

is an ASCII character.

Input Processing

In an input statement, the H field descriptor transfers n characters from the external field to the field descriptor. The first character appears immediately after the letter H. Any characters in the field descriptor before input are replaced by the input characters.

Output Processing

In an output statement, the H field descriptor transfers n characters following the letter H from the field descriptor to the external field.

12.2.9.3 Character Constants

You can use a character constant instead of an H field descriptor; both types of format specifier function identically.

In a character constant, the apostrophe is written as two apostrophes. For example:

```
50  FORMAT ('TODAY' 'S△DATE△IS:△',I2,'/',I2,'/',I2)
```

A pair of apostrophes used this way is considered a single character.

12.2.10 Default Field Descriptors

If you write the field descriptors I, O, Z, L, F, E, D, G, or A without specifying a field width value, default values for w, d, and e are supplied based on the data type of the I/O list element.

Table 12-2 lists the default values for w, d, and e.

Table 12-2: Default Field Descriptor Values

Field Descriptor	List Element	w	d	e
I,O,Z	BYTE	7		
I,O,Z	INTEGER*2,LOGICAL*2	7		
I,O,Z	INTEGER*2,LOGICAL*4	12		
O,Z	REAL*4	12		
O,Z	REAL*8	23		
O,Z	REAL*16	44		
L	LOGICAL	2		
F,E,G,D	REAL, COMPLEX*8	15	7	2
F,E,G,D	REAL*8, COMPLEX*16	25	16	2
F,E,G,D	REAL*16	42	33	3
A	LOGICAL*1	1		
A	LOGICAL*2,INTEGER*2	2		
A	LOGICAL*4,INTEGER*4	4		
A	REAL*4,COMPLEX*8	4		
A	REAL*8,COMPLEX*16	8		
A	REAL*16	16		
A	CHARACTER*n	n		

Note that for the A field descriptor, the default is the actual length of the corresponding I/O list element.

12.2.11 Positional Editing

Positional editing is controlled by the X, T, TL and TR edit descriptors.

On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transmitted and therefore does not by itself affect the length of the record. If characters are transmitted to positions at or after the position specified by a T, TL, TR, or X edit descriptor, positions skipped and not previously filled are filled with blanks. The result is as if the entire record were initially filled with blanks.

12.2.11.1 X Edit Descriptor

The X edit descriptor is a positional specifier. It has the form:

nX

The term n specifies how many character positions are to be passed over. The value of n must be greater than or equal to one.

Input Processing

In an input statement, the X field descriptor specifies that the next n characters in the input record are to be skipped.

Output Processing

In an output statement, the X field descriptor tabs right n spaces; it does not write over anything already written on the same record. For example:

```
WRITE (6,90) NPAGE
90  FORMAT ('1PAGE△NUMBER△',I2,16X,'GRAPHIC△ANALYSIS,△CONT.')
```

The preceding WRITE statement would print a record similar to:

```
PAGE NUMBER nn           GRAPHIC ANALYSIS, CONT.
```

The term nn is the current value of the variable NPAGE. The numeral 1 in the first character constant is not printed; it is used to advance the printer paper to the top of a new page. Section 12.3 describes printer carriage control.

Note that a trailing X format on a record will not write any characters unless it is followed by another field that does. For example:

```
WRITE (6,99) K
99  FORMAT ('△K=',I6,5X)
```

The preceding example will write a record of only 9 characters. To cause n trailing blanks to be written at the end of a record, use the format n('△').

12.2.11.2 T Edit Descriptor

The T edit descriptor is a positional tabulation specifier. It has the form:

Tn

The term *n* indicates the character position of the external record. The value of *n* must be greater than or equal to one.

Input Processing

In an input statement, the T field descriptor positions the external record to its *n*th character position. For example, if an input statement reads a record containing:

```
ABC△△△XYZ
```

and this record is under the control of the FORMAT statement:

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

on execution, the input statement would first read the characters XYZ and then read the characters ABC.

Output Processing

In an output statement, the T field descriptor specifies that subsequent data transfer is to begin at the *n*th character position of the external record. The first position of a record to be printed is usually reserved for a carriage control character, which is not printed (see Section 12.3). For example:

```
PRINT 25  
25  FORMAT (T51,'COLUMN 2',T21,'COLUMN 1')
```

These statements would print the following line (assuming normal carriage control processing):

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

12.2.11.3 TL Edit Descriptor

The TL edit descriptor is a relative tabulation specifier. It has the form:

```
TLn
```

The term *n* indicates that the next character to be transferred from or to a record is the *n*th character to the left of the current character. The value of *n* must be greater than or equal to one. If the value of *n* is greater than or equal to the current character position, the first character in the record is specified.

12.2.11.4 TR Edit Descriptor

The TR edit descriptor is also a relative tabulation specifier. It has the form:

```
TRn
```

The term *n* indicates that the next character to be transferred from or to a record is the *n*th character to the right of the current character. The value of *n* must be greater than or equal to one.

12.2.12 Miscellaneous Editing Operations

Edit descriptors that fall into the miscellaneous category are Q, dollar sign (\$), and colon (:).

- The Q edit descriptor obtains the number of characters remaining following a partial read operation.
- The \$ edit descriptor controls carriage returns.
- The : edit descriptor terminates format control if no more items are in the I/O list.

These descriptors are discussed, in this order, in the subsections that follow.

12.2.12.1 Q Edit Descriptor

The Q edit descriptor obtains the number of characters in the input record remaining to be transferred during a read operation. It has the form:

Q

The corresponding I/O list element must be of integer or logical data type.

For example:

```
      READ (4,1000) XRAY, KK, NCHRS, (ICHR(I), I=1,NCHRS)
1000 FORMAT (E15.7,I4,Q,80A1)
```

The preceding input statements read two fields into the variables XRAY and KK. The number of characters remaining in the record is stored in NCHRS, and exactly that many characters are read into the array ICHR. By placing the Q descriptor first in the format specification, you can determine the actual length of the input record.

In an output statement, the Q edit descriptor has no effect except that the corresponding I/O list element is skipped.

12.2.12.2 Dollar Sign Descriptor

The dollar sign character (\$) in a format specification modifies the carriage control specified by the first character of the record. It only affects those files for which the 'FORTRAN' carriage control attribute (see Section 12.3) is in effect.

In an input statement, the \$ descriptor is ignored.

In an output statement, if the first character of the record is a space, the \$ descriptor suppresses the carriage return. For terminal I/O, this means that a typed response will follow the output on the same line. If the first character of the record is a plus sign (+), the \$ descriptor causes the output to begin at the end of the previous line and leaves the print position at the end of the line. If the first character of the record is 0 or 1, the \$ descriptor is ignored.

Thus, the statements

```
TYPE 100
100 FORMAT (' ENTER RADIUS VALUE ', $)
ACCEPT 200, RADIUS
200 FORMAT (F6.2)
```

produce a message on the terminal in the form:

```
ENTER RADIUS VALUE
```

Your response (for example, "12.") can then go on the same line:

```
ENTER RADIUS VALUE 12.
```

12.2.12.3 Colon Descriptor

The colon character (:) in a format specification terminates format control if no more items are in the I/O list. The : descriptor has no effect if I/O list items remain. For example:

```
PRINT 1,3
PRINT 2,4
1 FORMAT (' I=', I2, ' J=', I2)
2 FORMAT (' K=', I2, ':', ' L=', I2)
```

These statements print the following two lines:

```
I=Δ3ΔJ=
K=Δ4
```

Section 12.7 describes format control in detail.

12.3 Carriage Control

Whenever the default for the OPEN statement's CARRIAGECONTROL keyword is in effect ('FORTRAN'), the first character of every record transferred to a printer is not printed. Instead, it is interpreted as a carriage control character (except when overridden by the OPEN statement keyword CARRIAGECONTROL = 'LIST' or 'NONE'). The I/O system recognizes certain characters as carriage control characters. Table 12-3 lists these characters and their effects.

Table 12-3: Carriage Control Characters

Character	Meaning
'+'	Overprinting: starts output at the beginning of the current line and returns to the left margin after printing
'Δ'	Single spacing: starts output at the beginning of the next line
'0'	Double spacing: skips a line before starting output

Table 12-3 (Cont.): Carriage Control Characters

Character	Meaning
'1'	Paging: starts output at the top of a new page
'\$'	Prompting: starts output at the beginning of the next line, and suppresses carriage return at the end of the line
ASCII NUL	Overprinting with no advance: starts output at the beginning of the current line and does not return to the left margin after printing

Any character other than those listed in Table 12-3 is treated as a space and is deleted from the print line. Note that if you accidentally omit the carriage control character, the first character of the record is not printed.

12.4 Format Specification Separators

Field descriptors in a format specification are generally separated by commas. You can also use the slash (/) record terminator to separate field descriptors. A slash terminates input or output of the current record and initiates a new record. For example:

```
WRITE (6,40) K,L,M,N,O,P
40  FORMAT (3I6,6/I6,2F8,4)
```

The preceding statements are equivalent to the following statements.

```
WRITE (6,40) K,L,M
40  FORMAT (3I6,6)
WRITE (6,50) N,O,P
50  FORMAT (I6,2F8,4)
```

You can use multiple slashes to bypass input records or to output blank records. If *n* consecutive slashes appear between two field descriptors, (*n*-1) records are skipped on input, or (*n*-1) blank records are output. The first slash terminates the current record; the second slash terminates the first skipped or blank record, and so on.

However, *n* slashes at the beginning or end of a format specification result in *n* skipped or blank records. This is because the opening and closing parentheses of the format specification are themselves a record initiator and terminator, respectively. For example:

```
WRITE (6,99)
99  FORMAT ('1',T51,'HEADING LINE'///T51,'SUBHEADING LINE'///)
```

The above statements produce the following output:

```
Column 50, top of page
      ↓
(blank line)  HEADING LINE
              SUBHEADING LINE
(blank line)
(blank line)
```

12.5 External Field Separators

A field descriptor such as Fw.d specifies that an input statement is to read w characters from the external record. If the data field in the external record contains fewer than w characters, the input statement would read characters from the next data field in the external record, unless the short field is padded with leading zeros or spaces.

When the field descriptor is numeric, you can avoid padding the input field by using a comma to terminate the field. The comma overrides the field descriptor's field width specification. This is called short field termination. It is particularly useful when you are entering data from a terminal keyboard. You can use it with the I, O, Z, F, E, D, G, and L field descriptors. For example:

```
      READ (5,100) I,J,A,B
100  FORMAT (2I6,2F10.2)
```

If the preceding statements read the following record:

1,-2,1.0,35

Based on this input, the following assignments would occur:

I = 1

J = -2

A = 1.0

B = 0.35

Note that the physical end of the record also serves as a field terminator and that the d part of a w.d specification is not affected by an external field separator.

You can use a comma to terminate only fields less than w characters long. If a comma follows a field of w or more characters, the comma is considered part of the next field.

Two successive commas, or a comma after a field of w characters, constitutes a null (zero-length) field. Depending on the field descriptor specified, the resulting value assigned is 0, 0.0, 0.D0, 0.Q0, or .FALSE..

You cannot use a comma to terminate a field that is controlled by an A, H, or character constant field descriptor. However, if the record reaches its physical end before w characters are read, short field termination occurs and the characters that were read are assigned successfully. Trailing spaces are appended to fill the corresponding I/O list element or the field descriptor.

12.6 Run-Time Format

You can store format specifications in character scalar references, numeric array references, numeric scalar field references (see Section 6.2.5.3), or numeric array references. Such a format specification is called a run-time format, and can be constructed or altered during program execution.

A run-time format in an array has the same form as a FORMAT statement, without the word FORMAT and the statement label. The opening and closing parentheses are required. Variable format expressions are not permitted in run-time formats.

In the following example, the DATA statement assigns a left parenthesis to the character array element FORCHR(0), and assigns a right parenthesis and three field descriptors to four character variables for later use. Next, the proper field descriptors are selected for inclusion in the format specification. The selection is based on the magnitude of the individual elements of the array TABLE. A right parenthesis is then added to the format specification just before the WRITE statement uses it. Thus, the format specification changes with each iteration of the DO loop.

```
SUBROUTINE PRINT(TABLE)
REAL TABLE(10,5)
CHARACTER*5 FORCHR(0:5), RPAR*1, FBIG, FMED, FSML
DATA FORCHR(0),RPAR /'(',')'/
DATA FBIG,FMED,FSML /'F8.2','F9.4','F9.6'/
DO 20 I=1,10
  DO 18 J=1,5
    IF (TABLE(I,J) .GE. 100.) THEN
      FORCHR(J) = FBIG
    ELSE IF (TABLE(I,J) .GT. 0.1) THEN
      FORCHR(J) = FMED
    ELSE
      FORCHR(J) = FSML
    END IF
18    CONTINUE
    FORCHR(5)(5:5) = RPAR
    WRITE (6,FORCHR) (TABLE(I,J), J=1,5)
20  CONTINUE
END
```

NOTE

Format specifications stored in arrays are recompiled at run time each time they are used. If a Hollerith or character run-time format is used in a READ statement to read data into the format itself, that data is not copied back into the original array. Thus, it will not be available subsequently for using that array as a run-time format specification.

12.7 Format Control Interaction with I/O Lists

Format control begins with the execution of a formatted I/O statement. The action taken by format control depends on information provided jointly by the next element of the I/O list (if one exists) and the next field descriptor of the format specification. Both the I/O list and the format specification are interpreted from left to right, except when repeat counts and implied-DO lists are specified.

If the I/O statement contains an I/O list, you must specify at least one I, O, Z, F, E, D, G, L, A, or Q field descriptor in the format specification. An error occurs if a field descriptor is not specified in this case.

On execution, a formatted input statement reads one record from the specified unit and initiates format control. Thereafter, additional records can be read as indicated by the format specification. Format control requires that a new record be read when a slash occurs in the format specification, or when the last closing parenthesis of the format specification is reached and I/O list elements remain to be filled. Any remaining characters in the current record are discarded when the new record is read.

On execution, a formatted output statement transmits a record to the specified unit as format control terminates. Records can also be written during format control if a slash appears in the format specification or if the last closing parenthesis is reached and more I/O list elements remain to be transferred.

The I, O, Z, F, E, D, G, L, A, and Q field descriptors each correspond to one element in the I/O list. No list element corresponds to an H, X, P, T, TL, TR, SP, SS, S, BN, BZ, \$, :, or character constant field descriptor. In H and character constant field descriptors, data transfer occurs directly between the external record and the format specification.

When an I/O list element is to be transferred, format field descriptors are processed, beginning with the current format item, until a descriptor is found that corresponds to an I/O list element. The I/O list element is then transferred under control of the field descriptor.

Format execution continues until one of the following is encountered: an element-transferring field descriptor; a colon (:) edit descriptor; or the end of the format. These also terminate format execution when no I/O list elements are to be transferred.

When the last closing parenthesis of the format specification is reached, format control determines whether more I/O list elements are to be processed. If not, format control terminates. However, if additional list elements remain, part or all of the format specification is reused in a process called format reversion.

In format reversion, the current record is terminated, a new one is initiated, and format control reverts to the group repeat specification whose opening parenthesis matches the next-to-last closing parenthesis of the format specification. If the format does not contain a group repeat specification, format control returns to the initial opening parenthesis of the format specification. Format control continues from that point.

12.8 Summary of Rules for FORMAT Statements

The following sections summarize the rules for constructing and using the format specifications and their components and for constructing external fields and records. Table 12-4 summarizes the FORMAT codes.

Table 12-4: Summary of FORMAT Codes

Code	Form	Effect
A	A[w]	Transfers character or Hollerith values.
BN	BN	Specifies that embedded and trailing blanks in a numeric input field are to be ignored.
BZ	BZ	Specifies that embedded and trailing blanks in a numeric input field are to be treated as zeros.
D	Dw.d	Transfers real values (D exponent field indicator).
E	Ew.d[Ee]	Transfers real values (E exponent field indicator).
F	Fw.d	Transfers real values.
G	Gw.d[Ee]	Transfers real values: on input, acts like F code; on output, acts like E code or F code, depending on the magnitude of the value.
H	nHc...c	Transfers data between the H field descriptor and an external record.
I	Iw[.m]	Transfers decimal integer values.
L	Lw	Transfers logical data: on input, transfers characters; on output, transfers T or F.
O	Ow[.m]	Transfers octal values.
Q	Q	Obtains the number of characters remaining to be transferred in an input record.
S	S	Reinvokes optional plus characters in numeric output fields: counters the action of SP and SS.
SP	SP	Writes plus characters that would otherwise be optional into numeric output fields.
SS	SS	Suppresses optional plus characters in numeric output fields.
T	Tn	Specifies positional tabulation.
TL	TLn	Specifies relative tabulation (left).
TR	TRn	Specifies relative tabulation (right).
X	nX	Specifies that n characters are to be skipped.
Z	Zw[.m]	Transfers hexadecimal values.
\$	\$	Suppresses carriage return during interactive I/O.
:	:	Terminates format control if the I/O list is exhausted.

12.8.1 General Rules

1. A FORMAT statement must always be labeled.
2. In a field descriptor such as rIw[m] or nX, the terms r, w, m, and n must be unsigned integer constants or variable format expressions whose values are greater than or equal to zero. The values of r and w must be greater than zero and less than or equal to 32767, and the values of m and n must be greater than zero and less than or equal to 255. (They cannot be names assigned to constants in PARAMETER statements.) You can omit the repeat count and field width specification.
3. In a field descriptor such as Fw.d, the term d must be an unsigned integer constant or variable format expression. You must specify d with F, E, D, and G field descriptors even if d is zero. The decimal point is also required. You must either specify both w and d, or omit them both. In a field descriptor such as Ew.dEe, the term e must also be an unsigned integer constant.
4. In a field descriptor such as nHc₁c₂ ... c_n, exactly n characters must follow the H format code. You can use any printing ASCII character in this field descriptor.
5. In a scale factor of the form nP, n must be an integer constant or variable format expression in the range -128 through 127 inclusive. The scale factor affects the F, E, D, and G field descriptors only. Once you specify a scale factor, it applies to all subsequent real field descriptors in that format specification until another scale factor appears. You must explicitly specify 0P to reinstate a scale factor of zero. Format reversion does not affect the scale factor.
6. No repeat count is permitted in BN, BZ, S, SS, SP, H, Q, X, T, TR, TL, \$, :, or character constant field descriptors unless these descriptors are enclosed in parentheses and treated as a group repeat specification.
7. If the associated I/O statement contains an I/O list, the format specification must contain at least one field descriptor. This descriptor must be I, O, Z, F, E, D, G, L, A, or Q.
8. A format specification in a character variable, character substring reference, character array element, character array, character expression, numeric array, or numeric array element must be constructed in the same way as a format specification in a FORMAT statement, including the opening and closing parentheses.
9. If a character-constant format includes apostrophes, those apostrophes must be represented by double apostrophes.

12.8.2 Input Rules

1. A minus sign must precede a negative value in an external input field; a plus sign is optional before a positive value.
2. On input, an external field under I field descriptor control must be an integer constant. It cannot contain a decimal point or an exponent. An external field under O field descriptor control must contain only the numerals 0 through 7. An external field input under Z field descriptor control must contain only the numerals 0 through 9 and the letters A(a) through F(f). An external field under O or Z field descriptor control must not contain a sign, a decimal point, or an exponent. You cannot use octal and hexadecimal constants in the form '777'O or 'AF9'X in external records.
3. On input, an external field under F, E, D, or G field descriptor control must be an integer constant or a real constant. It can contain a decimal point and/or an E(e), D(d), or Q(q) exponent field.
4. If an external field contains a decimal point, the actual size of the fractional part of the field, as indicated by that decimal point, overrides the d specification of the corresponding real field descriptor.
5. If an external field contains an exponent, the scale factor (if any) of the corresponding field descriptor is inoperative for the conversion of that field.
6. The field width specification must be large enough to accommodate both the numeric character string of the external field and any other characters that are allowed (algebraic sign, decimal point, and/or exponent).
7. A comma is the only character you can use as an external field separator. It terminates the input of fields (for noncharacter data types) that are shorter than the number of characters expected. It also designates null (zero-length) fields.

12.8.3 Output Rules

1. A format specification cannot specify more output characters than the external record can contain. For example, a line printer record cannot contain more than 133 characters, including the carriage control character.
2. The field width specification (w) must be large enough to accommodate all characters that the data transfer can generate, including an algebraic sign, decimal point, and exponent. For example, the field width specification in an E field descriptor should be large enough to contain d+7 or d+e+5 characters.
3. The first character of a record transmitted to a line printer or terminal is typically used for carriage control; it is not printed. The first character of such a record should be a space, 0, 1, \$, +, or ASCII NUL. Any other character is treated as a space.

Chapter 13

Auxiliary Input/Output Statements

The auxiliary I/O statements perform file management functions. These statements are:

- **OPEN**—associates FORTRAN logical units with files. **OPEN** establishes a connection between a logical unit and a file or device, and declares the attributes required for read and write operations.
- **CLOSE**—terminates the connection between a logical unit and a file or device.
- **INQUIRE**—inquires about specified properties of a file or a logical unit.
- **REWIND** and **BACKSPACE**—perform file-positioning functions.
- **ENDFILE**—writes a special form of record that causes an end-of-file condition (and **END=** transfer) when an input statement reads the record.
- **DELETE**—deletes a record from a file.
- **UNLOCK**—unlocks a currently accessed record, permitting access by other programs.

The statements are described, in the order shown here, in the sections that follow.

13.1 OPEN Statement

An **OPEN** statement either connects an existing file to a logical unit or creates a new file and connects it to a logical unit. In addition, **OPEN** can specify file attributes that control file creation and/or subsequent processing.

The **OPEN** statement has the form:

```
OPEN (par[,par]...)
```

where:

par

is a keyword specification in one of the following forms:

```
keywd
```

```
keywd = value
```


where:

keywd

is a keyword, as described below.

value

depends on the keyword, as described below.

Keywords are divided into several categories based on function:

- Keywords that identify the unit and the file:

UNIT	—logical unit number to be used
FILE or NAME	—file-name specification for the file
DEFAULTFILE	—default file-name specification for the file
or TYPE	—file existence status at OPEN
DISPOSE	—file existence status after CLOSE

- Keywords that describe the file processing to be performed:

ACCESS	—FORTRAN access method to be used
ORGANIZATION	—logical file structure
READONLY	—write protection

- Keywords that describe the records in the file:

BLOCKSIZE	—physical block size
CARRIAGECONTROL	—printer control type
FORM	—type of FORTRAN record formatting
RECL or RECORDSIZE	—logical record length
RECORDTYPE	—logical record format
BLANK	—blank interpretation for numeric input
KEY	—positions of key fields within records in an indexed file

- Keywords that describe file storage allocation when a file is created:

INITIALSIZE	—initial file allocation
EXTENDSIZE	—file allocation increment size

- Keywords that provide additional capability for direct access I/O:

ASSOCIATEVARIABLE	—the next record number value
MAXREC	—maximum direct access record number

- Optional keywords that provide improved performance or special capabilities. These options are generally transparent to I/O processing:

BUFFERCOUNT	—number of I/O buffers to be used
NOSPANBLOCKS	—records are not to be split across physical blocks
SHARED	—other programs can simultaneously access the file
USEROPEN	—user program option to provide additional OPEN capability

ERR —statement to which control is transferred if an error occurs during execution of the OPEN statement

Iostat —status value that indicates whether an error condition exists

Table 13-1 lists the values accepted for each keyword.

Table 13-1: OPEN Statement Keyword Values

Keyword	Values ¹	Function	Default
ACCESS	'SEQUENTIAL' 'DIRECT' 'KEYED' 'APPEND'	Access mode	'SEQUENTIAL'
ASSOCIATEVARIABLE	asv	Next direct access record	
BLANK	'NULL' 'ZERO'	Interpretation of blanks	'NULL'
BLOCKSIZE	e	Physical block size	System default
BUFFERCOUNT	e	Number of I/O buffers	System default
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'	Print control	'FORTRAN' (formatted) 'NONE' (unformatted)
DEFAULTFILE	cl	Default file specification	
DISPOSE DISP	'KEEP' or 'SAVE' 'DELETE' 'PRINT' 'PRINT'/DELETE' 'SUBMIT' 'SUBMIT/DELETE'	File disposition at close	'KEEP'
ERR	s	Error transfer label	
EXTENDSIZE	e	File allocation increment	Volume or system default
FORM	'FORMATTED' 'UNFORMATTED'	Format type	Depends on ACCESS keyword

Table 13-1 (Cont.): OPEN Statement Keyword Values

Keyword	Values ¹	Function	Default
FILE NAME	c	File-name specifica- tion	
INITIALSIZE	e	File allocation	
IOSTAT	v	I/O status	
KEY	e1:e2[:INTEGER] [:CHARACTER]	Key field definitions	
MAXREC	e	Direct access record limit	
NOSPANBLOCKS	—	Records do not span blocks	
ORGANIZATION	'SEQUENTIAL' 'RELATIVE' 'INDEXED'	File structure	'SEQUENTIAL'
READONLY	—	Write protection	
RECL RECORDSIZE	e	Record length	As specified at file creation
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED' 'STREAM' 'STREAM_CR' 'STREAM_LF'	Record structure	Depends on ORGANIZATION, ACCESS, and FORM keywords
SHARED	—	File sharing allowed	
STATUS TYPE	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'	File status at open	'UNKNOWN'
UNIT	e	Logical unit number	
USEROPEN	p	User program option	

¹ Key: v is an integer scalar memory reference.
e is a numeric expression.
s is a statement label.
c is a character scalar reference, numeric scalar memory reference, or numeric array name reference.
c1 is a character expression.
e1 is the first byte position of a key.
e2 is the last byte position of a key.
p is an external function.

You can specify character values at run time by substituting a general character expression for a keyword value in the OPEN statement. The character value may contain trailing spaces, but it must not contain either leading or embedded spaces. For example:

```
CHARACTER*7 QUAL '//'  
:  
:  
:  
IF (exp) QUAL = '/DELETE'  
OPEN (UNIT=1, STATUS='NEW', DISP='SUBMIT'//QUAL)
```

Keyword specifications can appear in any order. In most cases, they are optional; default values are provided in their absence. If the logical unit specifier is the first parameter in the list, the UNIT keyword is optional.

The following examples illustrate various uses of the OPEN statement.

1. The following statement creates a new sequential formatted file on unit 1 with the default file name FOR001.DAT.

```
OPEN (UNIT=1, STATUS='NEW', ERR=100)
```

2. The following statement creates a 50-block direct access file for temporary storage. The file is deleted at program termination.

```
OPEN (UNIT=3, STATUS='SCRATCH', ACCESS='DIRECT',  
1 INITIALSIZE=50, RECL=64)
```

3. The following statement creates a file on magnetic tape with a large block size for efficient processing.

```
OPEN (UNIT=1, FILE='MTA0:MYDATA.DAT', BLOCKSIZE=8192,  
1 STATUS='NEW', ERR=14, RECL=1024,  
1 RECORDTYPE='FIXED')
```

4. The following statement opens the file created in the previous example for input.

```
OPEN (UNIT=1, FILE='MTA0:MYDATA.DAT', READONLY,  
1 STATUS='OLD', RECL=1024, RECORDTYPE='FIXED',  
1 BLOCKSIZE=8192)
```

5. The following statement uses the file name supplied by the user and the default file specification supplied by the DEFAULTFILE keyword to define the file specification for an existing file.

```
TYPE *, 'ENTER NAME OF DOCUMENT'  
ACCEPT *, DOC  
OPEN (UNIT=1, FILE=DOC, DEFAULTFILE='[ARCHIVE].TXT',  
1 STATUS='OLD')
```

Sections 13.1 through 13.1.28 describe in detail the parameters represented by the various keywords. As used in these sections, a numeric expression can be any integer or real expression. The value of the expression is converted to integer data type before it is used in the OPEN statement.

13.1.1 ACCESS Keyword

The ACCESS parameter has the form:

```
ACCESS = acc
```

where:

acc

is a character expression having a value equal to 'DIRECT', 'SEQUENTIAL', 'KEYED', or 'APPEND'.

ACCESS specifies whether the file is to be opened for keyed, direct, or sequential access. If you specify 'DIRECT', the file is accessed by record number. If you specify 'SEQUENTIAL', the file is accessed sequentially. If you specify 'KEYED', the file is accessed by a specified key. 'APPEND' implies sequential access and positioning after the last record of the file. The default is 'SEQUENTIAL'.

13.1.2 ASSOCIATEVARIABLE Keyword

The ASSOCIATEVARIABLE parameter has the form:

```
ASSOCIATEVARIABLE = asv
```

where:

asv

is an integer variable. It cannot be a dummy argument to the routine in which the OPEN statement appears.

ASSOCIATEVARIABLE specifies the integer variable that is updated after each direct access I/O operation to reflect the record number of the next sequential record in the file. This specifier is valid only for direct access mode; it is ignored for other access modes.

13.1.3 BLANK Keyword

The BLANK parameter has the form:

```
BLANK = blk
```

where:

blk

is a character expression having a value equal to either 'NULL' or 'ZERO'.

When BLANK specifies 'NULL', all blanks in a numeric input field are ignored (except if the field is comprised of all blanks, in which case it is treated as zero). When BLANK specifies 'ZERO', all blanks other than leading blanks are to be treated as zeros. The default value is 'NULL'.

If the /NOF77 compiler command qualifier is specified, the default value is 'ZERO'.

13.1.4 BLOCKSIZE Keyword

The BLOCKSIZE keyword specifies the physical I/O transfer size for the file. It has the form:

BLOCKSIZE = bks

where:

bks

is a numeric expression.

For magnetic tape files, the value of bks specifies the physical record size in the range 18 to 32767 bytes. The default value is 2048 bytes.

For sequential disk files, the value of bks is rounded up to an integral number of 512-byte blocks and used to specify RMS multiblock transfers. The number of blocks transferred can be 1 to 127. The number of blocks transferred is determined by RMS defaults. Refer to the description of the SET RMS_DEFAULT command in the *Guide to Using DCL and Command Procedures on VAX/VMS* for more information on setting process and system default multiblock counts if you do not specify a block size.

For relative files and indexed files, the value of bks is rounded up to an integral number of 512-byte blocks and used to specify the RMS bucket size in the range 1 to 32 blocks. The default is the smallest value capable of holding a single record.

13.1.5 BUFFERCOUNT Keyword

The BUFFERCOUNT parameter has the form:

BUFFERCOUNT = bc

where:

bc

is a numeric expression.

The range of values for bc is from 1 to 127. The size of each buffer is determined by the BLOCKSIZE keyword. Thus, if BUFFERCOUNT=3 and BLOCKSIZE=2048, the total number of bytes allocated for buffers is 3*2048, or 6144.

BUFFERCOUNT specifies the number of buffers to be associated with the logical unit for multibuffered I/O. The BLOCKSIZE keyword determines the size of each buffer. If you do not specify BUFFERCOUNT, or if you specify zero, the system default is assumed. Refer to the description of the SET RMS_DEFAULT command in the *Guide to Using DCL and Command Procedures on VAX/VMS* for information on setting process and system default buffer counts.

13.1.6 CARRIAGECONTROL Keyword

The CARRIAGECONTROL parameter has the form:

```
CARRIAGECONTROL = cc
```

where:

cc

is a character expression having a value equal to 'FORTRAN', 'LIST', or 'NONE'.

The CARRIAGECONTROL parameter determines the type of carriage control processing to be used when printing a file. The default for formatted files is 'FORTRAN'; for unformatted files, the default is 'NONE'. 'FORTRAN' specifies normal FORTRAN interpretation of the first character, 'LIST' specifies single spacing between records, and 'NONE' specifies no implied carriage control.

13.1.7 DEFAULTFILE Keyword

The DEFAULTFILE parameter has the form:

```
DEFAULTFILE = ce
```

where:

ce

is a character expression which contains a default file name specification string.

The DEFAULTFILE keyword specifies a default file specification string. You can use this keyword to supply a value to the RMS default file specification string for the missing components of a file specification. If you do not specify the DEFAULTFILE keyword, FORTRAN uses the default value 'FORnnn.DAT', where nnn is the unit number with leading zero(s).

The default file specification string is used primarily when accepting file specifications interactively; file specifications known to a user program are normally completely specified in the FILE keyword. You can specify defaults for one or more of the following file specification components:

- node
- device
- directory
- filename
- filetype
- file version number

When you specify any of the above components in the FILE=keyword, they override those values specified in the DEFAULTFILE=keyword. Refer to the *VAX Record Management Services Reference Manual* for more information.

13.1.8 DISPOSE Keyword

The DISPOSE (or DISP) parameter has the form:

```
DISPOSE = dis
DISP = dis
```

where:

dis

is a character expression having a value equal to 'KEEP', 'SAVE', 'DELETE', 'PRINT', 'SUBMIT', 'PRINT/DELETE', or 'SUBMIT/DELETE'.

The DISPOSE parameter determines the disposition of the file connected to the unit when the unit is closed.

- If you specify 'KEEP' or 'SAVE', the file is retained after the unit is closed; this is the default value.
- If you specify 'DELETE', the file is deleted.
- If you specify 'PRINT', the file is submitted to the system line printer spooler and is not deleted; it is printed and then deleted if you specify 'PRINT/DELETE'.
- If you specify 'SUBMIT', the file is submitted to the batch job queue and is not deleted; it is submitted and then deleted if you specify 'SUBMIT/DELETE'.

A read-only file cannot be deleted. A scratch file cannot be saved, printed, or submitted.

13.1.9 ERR Keyword

The ERR parameter has the form:

```
ERR = s
```

where:

s

is the label of an executable statement.

The ERR parameter identifies the executable statement that is to receive control when an error occurs. ERR applies only to the OPEN statement in which it is specified, and not to subsequent I/O operations on the unit. If an error occurs, no file is opened or created.

13.1.10 EXTENDSIZE Keyword

The EXTENDSIZE parameter has the form:

EXTENDSIZE = es

where:

es

is a numeric expression.

The EXTENDSIZE parameter specifies the number of blocks by which to extend a disk file when additional storage space is allocated. If you do not specify EXTENDSIZE or if you specify zero, the system default for the device is used.

See Section 13.1.13 for a discussion about the relationship between the EXTENDSIZE keyword and the INITIALSIZE keyword.

13.1.11 FILE Keyword

The FILE parameter has the form:

FILE = fln

where:

fln

is a character scalar reference, numeric scalar memory reference, or numeric array name reference.

The FILE parameter specifies the name of the file to be connected to the unit. The name can be any file specification accepted by the operating system. Section 1.5.1 describes default file name conventions.

If the file name is stored in a numeric scalar or array, the name must consist of ASCII characters terminated by an ASCII null character (zero byte). However, if it is stored in a character scalar or array, it must not contain a zero byte.

13.1.12 FORM Keyword

The FORM parameter has the form:

FORM = ft

where:

ft

is a character expression having a value equal to 'FORMATTED' or 'UNFORMATTED'.

The **FORM** parameter specifies whether the file being opened is to be read or written using formatted or unformatted **READ** or **WRITE** statements. For sequential access files, 'FORMATTED' is the default. For direct access and keyed access files, 'UNFORMATTED' is the default.

13.1.13 INITIALSIZE Keyword

The **INITIALSIZE** parameter has the form:

```
INITIALSIZE = insz
```

where:

insz

is a numeric expression.

If you do not specify **INITIALSIZE**, or if you specify zero, no initial allocation is made. The system attempts to allocate contiguous space for **INITIALSIZE**. If not enough contiguous space is available, noncontiguous space is allocated.

The **INITIALSIZE** parameter specifies the number of blocks in the initial storage allocation for a disk file. The **EXTENDSIZE** parameter specifies the number of blocks by which a disk file is extended each time more space is needed for the file.

INITIALSIZE is effective only at the time the file is created. If **EXTENDSIZE** is specified when the file is created, the value specified is the default value used to allocate additional storage for the file. If you specify **EXTENDSIZE** when you open an existing file, the value you specify supersedes any **EXTENDSIZE** value specified when the file was created, and remains in effect until you close the file. Unless specifically overridden, the default **EXTENDSIZE** value is in effect on subsequent openings of the file.

13.1.14 IOSTAT Keyword

The **IOSTAT** parameter has the form:

```
IOSTAT = ios
```

where:

ios

is an integer scalar memory reference.

The **IOSTAT** parameter is an I/O status specifier. It causes **ios** to be defined as zero if no error condition exists, or as a positive integer if an error condition exists. **VAX FORTRAN** I/O status values are described in Sections 18.1 and 18.3. **IOSTAT** applies only to the **OPEN** statement in which it appears and not to subsequent I/O operations on the logical unit that is opened; however, **IOSTAT** can be used in subsequent I/O statements to perform a similar function (see Chapter 11).

13.1.15 KEY Keyword

The KEY parameter has the form:

KEY = (kspec[,kspec]...)

where:

kspec

has the form:

e1:e2[:dt]

where:

e1

is the first byte position of the key.

e2

is the last byte position of the key.

dt

is the data type of the key: either INTEGER or CHARACTER.

The KEY parameter defines the access keys for records in an indexed file. The key starts at position e1 in the record and has a length of e2-e1+1. The values of e1 and e2 must be such that:

1 .LE. (e1) .LE. (e2) .LE. record-length

1 .LE. (e2-e1+1) .LE. 255

If the key type is INTEGER, the key length must be either 2 or 4.

You must define at least one key for an indexed file. This mandatory key is called the primary key of the file and usually has a unique value for each record (this is the default condition). You can also define other keys, called alternate keys. RMS allows up to 254 alternate keys. The maximum allowed in an OPEN statement is, however, 85 and the use of other OPEN keywords reduces this limit further. See the section on FOR\$OPEN in the *VAX/VMS Run-Time Library Reference Manual* for more information on OPEN statement limits. A file that requires more keys than the practical limit for the OPEN statement must be created from another language or with FDL (File Definition Language). For information on FDL, see the *VAX Record Management Services Reference Manual*.

The default data type of a key is CHARACTER. The position of a key specification in the list determines a key's key-of-reference number. This number is used in any subsequent I/O statement to specify the same key. The primary key is key-of-reference number 0, the first alternate key is key-of-reference number 1, and so forth.

The key fields and key-of-reference numbers are permanent attributes of an indexed file and are established when the file is created. The KEY parameter must be specified when a file is created, but does not need to be specified when an existing file is opened. When an

existing file is opened, key definitions and key-of-reference numbers are obtained from the file itself. If the KEY parameter is specified for an existing file, it must agree with the established attributes of the file.

13.1.16 MAXREC Keyword

The MAXREC parameter has the form:

MAXREC = mr

where:

mr

is a numeric expression.

The MAXREC parameter specifies the maximum number of records permitted in a direct access file. The default is an unlimited number of records. This specifier applies only to direct access files.

13.1.17 NAME Keyword

NAME is a nonstandard synonym for FILE. See Section 13.1.11.

13.1.18 NOSPANBLOCKS Keyword

The NOSPANBLOCKS parameter has the form:

NOSPANBLOCKS

The NOSPANBLOCKS parameter specifies that records are not to cross disk block boundaries. If any record exceeds the size of a physical block, an error occurs.

13.1.19 ORGANIZATION Keyword

The ORGANIZATION parameter has the form:

ORGANIZATION = org

where:

org

is a character expression whose value is equal to 'SEQUENTIAL', 'RELATIVE', or 'INDEXED'.

The ORGANIZATION parameter specifies the internal organization of the file. The default file organization is sequential. However, if you omit the ORGANIZATION keyword when you open an existing file, the organization already specified in that file is used. If you specify ORGANIZATION for an existing file, org must have the same value as that of the existing file.

13.1.20 READONLY Keyword

The READONLY parameter has the form:

READONLY

The READONLY parameter specifies that an existing file can be read, but prohibits writing to that file.

The FORTRAN I/O system's default file access privileges are read-write, which can cause run-time I/O errors if the file protection does not permit write access. The READONLY keyword has no effect on the protection specified for a file. Its main purpose is to allow a file to be read simultaneously by two or more programs. For example, if you wish to open a file for the purpose of reading the file but want to allow others to read the same file while you have it open, specify the READONLY keyword. Refer to the *VAX FORTRAN User's Guide* for information on file sharing.

13.1.21 RECL Keyword

The RECL parameter has the form:

RECL = rl

where:

rl

is a numeric expression indicating the length of logical records in the file.

The value of rl does not include space for control information, such as for two segment control bytes (if present) or the bytes that RMS requires for maintaining record length and deleted record control information. The specification is for record data only.

The value of rl is interpreted as either bytes or longwords, depending on whether the records are formatted (bytes) or unformatted (longwords, that is, 4-byte units). Table 13-2 summarizes the maximum values that can be specified for rl, based on file organization and record format.

Table 13-2: Record Size (RECL) Limits

File Organization	Record Format	
	Formatted (bytes)	Unformatted (longwords)
Sequential	32766	8191
Sequential and variable-length records on ANSI magnetic tape	9999 ¹	2499 ¹
Relative and indexed	16380	4095

¹ Limit imposed by 4-byte ASCII count field.

The interpretation and effect of the logical record length varies under the following conditions:

- If the file contains fixed-length records, RECL specifies the size of each record.
- If the file contains variable-length records, RECL specifies the maximum length for any record.
- If the records are formatted, the length is the number of bytes.
- If the records are unformatted, the length is the number of longwords.
- If you omit this specifier for existing files, the record length specified when the file was created is assumed.
- If your program attempts to write, to an existing file, a record that is longer than the logical record length, an error is signaled.

If you are opening an existing file that contains fixed-length records or that has relative organization, and you specify a value for RECL that is different from the actual length of the records in the file, an error occurs. If you omit RECL when opening an existing file, the record length specified when the file was created is used.

You must specify RECL when you create files with fixed-length records or with relative or indexed organization.

13.1.22 RECORDSIZE Keyword

RECORDSIZE is a nonstandard synonym for RECL; refer to Section 13.1.21 for more information.

13.1.23 RECORDTYPE Keyword

The RECORDTYPE parameter has the form:

```
RECORDTYPE = typ
```

where:

typ

is a character expression whose value is equal to 'FIXED', 'VARIABLE', 'SEGMENTED', 'STREAM', 'STREAM_CR', or 'STREAM_LF'.

The RECORDTYPE parameter specifies whether the file has fixed-length records, variable-length records, segmented records, or stream-type variable-length records. When you create a file, the default record types are:

File Type	Default Record Type
Relative or indexed files	'FIXED'
Direct access sequential files	'FIXED'
Formatted sequential access files	'VARIABLE'
Unformatted sequential access files	'SEGMENTED'

A segmented record consists of one or more variable-length records. Use of segmented records allows a FORTRAN logical record to span several RMS records. Only unformatted sequential access files with sequential organization can use segmented records. You cannot specify 'SEGMENTED' for any other file type.

If you do not specify the RECORDTYPE parameter when you are accessing an existing file, the record type of the file is used. An exception to this rule is unformatted sequential access files with sequential organization and variable-length records; these files have a default of 'SEGMENTED'.

If you do specify the RECORDTYPE parameter when you are accessing an existing file, the type that you specify must match the type of an existing file.

In fixed-length record files, if an output statement does not specify a full record, the record is filled with spaces (for a formatted file) or zeros (for an unformatted file).

You cannot use an unformatted READ statement to access an unformatted sequential organization file containing variable-length records, unless you specify the corresponding RECORDTYPE value in your OPEN statement.

Files containing segmented records can be accessed only by unformatted sequential FORTRAN I/O statements.

13.1.24 SHARED Keyword

The SHARED parameter has the form:

SHARED

The SHARED parameter specifies that the file can be opened for shared access by more than one program executing simultaneously.

See the section on "file sharing" in the *VAX FORTRAN User's Guide* for additional information on this keyword.

13.1.25 STATUS Keyword

The STATUS parameter has the form:

STATUS = sta

where:

sta

is a character expression whose value is equal to 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'.

The STATUS parameter specifies the status of the file to be opened.

- If you specify 'OLD', the file must already exist.
- If you specify 'NEW', a new file is created.

- If you specify 'SCRATCH', a new file is created and it is deleted when the file is closed.
- If you specify 'UNKNOWN', the processor will first try 'OLD'; if the file is not found, the processor will use 'NEW', thereby creating a new file.

The default is 'UNKNOWN'.

If the /NOF77 compiler command qualifier is specified, the default value is 'NEW'.

NOTE

The STATUS parameter is also used in CLOSE statements to specify the status of a file after the file is closed; however, the values it uses are different from those used in OPEN statements.

13.1.26 TYPE Keyword

TYPE is a nonstandard synonym for STATUS. See Section 13.1.25.

13.1.27 UNIT Keyword

The UNIT parameter has the form:

[UNIT=] u

where:

u

is a numeric expression.

The UNIT parameter specifies the logical unit to which a file is to be connected. The unit specification must appear in the list. The UNIT keyword can be omitted only when the unit specifier occupies the first position in the list.

The logical unit may already be connected to a file when an OPEN statement is executed. If this file is not the same as the one to be opened, the OPEN statement executes as if a CLOSE statement had executed just before it. If the file to be opened is already connected to the unit or if the file specifier (FILE keyword) is not included in the OPEN statement, only the blank specifier (BLANK keyword) can have a value different from the one currently in effect. The position of the file is unaffected.

13.1.28 USEROPEN Keyword

The USEROPEN parameter has the form:

USEROPEN = procedure-name

where:

procedure-name

is the symbolic name of the USEROPEN procedure.

The procedure name must be declared **EXTERNAL**.

The **USEROPEN** parameter specifies a user-written external function that controls the opening of the file. Knowledgeable users can employ additional features of the operating system that are not directly available from **FORTRAN**, while retaining the convenience of writing programs in **FORTRAN**. See the *VAX FORTRAN User's Guide* for more information on **USEROPEN**.

13.2 CLOSE Statement

The **CLOSE** statement disconnects a file from a unit. It has the form:

$$\text{CLOSE } ([\text{UNIT}=\text{u}], \left\{ \begin{array}{l} \text{STATUS} \\ \text{DISPOSE} \\ \text{DISP} \end{array} \right\} = \text{p} \quad [,\text{ERR}=\text{s}][,\text{IOSTAT}=\text{ios}]$$

where:

u

is a logical unit number.

p

is a character expression that determines the disposition of the file. Its values are 'KEEP', 'SAVE', 'DELETE', 'PRINT', 'SUBMIT', 'SUBMIT/DELETE', and 'PRINT/DELETE'.

s

is the label of an executable statement.

ios

is an integer scalar memory reference.

The **CLOSE** statement parameters can occur in any order. The keyword **UNIT** is optional only if the unit specifier is the first parameter in the list.

If you specify either 'SAVE' or 'KEEP', the file is retained after the unit is closed. If you specify 'DELETE', the file is deleted. If you specify 'PRINT', the file is submitted to the line printer spooler; it is printed and deleted if you specify 'PRINT/DELETE'. If you specify 'SUBMIT', the file is submitted to the batch job queue; it is submitted and deleted if you specify 'SUBMIT/DELETE'. For scratch files, the default is 'DELETE'; for all other files, the default is 'KEEP'. The disposition specified in a **CLOSE** statement supersedes the disposition specified in the **OPEN** statement, except that a file opened as a scratch file cannot be saved, printed, or submitted and a file opened for read-only access cannot be deleted.

For example:

```
CLOSE (UNIT=1, STATUS='PRINT')
```

This statement closes the file on unit 1 and submits the file for printing.

```
CLOSE (UNIT=J, STATUS='DELETE', ERR=99)
```

This statement closes the file on unit J and deletes it.

13.3 INQUIRE Statement

The INQUIRE statement inquires about specified properties of a file or of a logical unit on which a file might be opened. The INQUIRE statement has two forms, one for inquiring by file and the other for inquiring by unit:

```
INQUIRE (FILE=fi[,DEFAULTFILE=dfi...],flist)
```

```
INQUIRE ([UNIT=u],flist)
```

where:

fi

is a character expression, numeric scalar memory reference, or numeric array name reference whose value specifies the name of the file to be inquired about.

dfi

is a character expression specifying a default file name specification string.

flist

is a list of property specifiers in which any one specifier appears only once. The specifiers are described in Sections 13.3.1 through 13.3.20.

u

is the number of the logical unit to be inquired about. The unit does not have to exist, nor does it need to be connected to a file. If the unit is connected to a file, the inquiry encompasses both the connection and the file.

FILE=fi and UNIT=u can appear anywhere in the property-specifier list; however, if the UNIT keyword is omitted, the unit specifier (u) must be the first parameter in the list.

DEFAULTFILE=dfi can be used in addition to or in place of FILE=fi when used in connection with an inquiry about a file. If a file is opened with both FILE and DEFAULT-FILE keywords specified in the OPEN statement, then you can inquire about this file by specifying both the FILE and DEFAULTFILE keywords in the INQUIRE statement.

An INQUIRE statement may be executed before, during, or after the connection of a file to a unit; the values assigned by the statement are those that are current when the INQUIRE statement is executed.

13.3.1 ACCESS Specifier

The ACCESS specifier has the form:

ACCESS = acc

where:

acc

is a character scalar memory reference.

Acc is assigned the value SEQUENTIAL if the file is connected for sequential access, DIRECT if the file is connected for direct access, and KEYED if the file is connected for keyed access. If there is no connection, acc is UNKNOWN.

13.3.2 BLANK Specifier

The BLANK specifier has the form:

BLANK = blk

where:

blk

is a character scalar memory reference.

Blk is assigned the value NULL if null blank control is in effect for a file connected for formatted I/O; it is assigned the value ZERO if zero blank control is in effect. If there is no connection or if the connection is not for formatted I/O, blk is assigned the value UNKNOWN.

13.3.3 CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier has the form:

CARRIAGECONTROL = cc

where:

cc

is a character scalar memory reference.

Cc is assigned the value FORTRAN if the file has the FORTRAN carriage control attribute, LIST if the file has the implied carriage control attribute, NONE if the file has no carriage control attribute, and UNKNOWN if no other value applies.

13.3.4 DIRECT Specifier

The DIRECT specifier has the form:

DIRECT = dir

where:

dir

is a character scalar memory reference.

Dir is assigned the value YES if DIRECT is an allowed access method for the file, NO if DIRECT is not an allowed access method, and UNKNOWN if the processor is unable to determine whether DIRECT is an allowed access method.

13.3.5 ERR Specifier

The ERR specifier has the form:

ERR = s

where:

s

is the label of an executable statement.

ERR is a control specifier rather than a property specifier. If an error occurs during execution of the INQUIRE statement, control is transferred to the statement whose label is s.

13.3.6 EXIST Specifier

The EXIST specifier has the form:

EXIST = ex

where:

ex

is a logical scalar memory reference.

Ex is assigned the value .TRUE. if the specified file or unit exists, and the value .FALSE. if the specified file or unit does not exist.

13.3.7 FORM Specifier

The FORM specifier has the form:

FORM = fm

where:

fm

is a character scalar memory reference.

Fm is assigned the value FORMATTED if the file is connected for formatted I/O, and UNFORMATTED if the file is connected for unformatted I/O. If there is no connection, fm is assigned the value UNKNOWN.

13.3.8 FORMATTED Specifier

The FORMATTED specifier has the form:

FORMATTED = fmd

where:

fmd

is a character scalar memory reference.

Fmd is assigned the value YES if formatted is an allowed form for the file, NO if formatted is not an allowed form, and UNKNOWN if the processor is unable to determine whether formatted is an allowed form.

13.3.9 IOSTAT Specifier

The IOSTAT specifier has the form:

IOSTAT = ios

where:

ios

is an integer scalar memory reference.

IOSTAT is a control specifier rather than a property specifier. Ios is assigned a processor-dependent positive integer value if an error occurs during execution of the INQUIRE statement; it is assigned the value ZERO if there is no error condition.

13.3.10 KEYED Specifier

The KEYED specifier has the form:

KEYED = kyd

where:

kyd

is a character scalar memory reference.

Kyd is assigned the value YES if KEYED is an allowed access method for the file (that is, the file is indexed), NO if KEYED is not an allowed access method, and UNKNOWN if the processor is unable to determine whether KEYED is an allowed access method.

13.3.11 NAME Specifier

The NAME specifier has the form:

NAME = nme

where:

nme

is a character scalar memory reference.

Nme is assigned the name of the file being inquired about. If the file does not have a name, nme is not defined.

The value assigned to nme is not necessarily identical to the value specified with the FILE keyword. For example, the value that the processor returns may be qualified by a directory name or a version number. However, the value that is assigned is always valid for use with the FILE keyword in an OPEN statement.

NOTE

FILE and NAME are synonyms when used with the OPEN statement, but not when used with the INQUIRE statement.

13.3.12 NAMED Specifier

The NAMED specifier has the form:

NAMED = nmd

where:

nmd

is a logical scalar memory reference.

Nmd is assigned the value .TRUE. if the specified file has a name, and the value .FALSE. if it does not have a name.

13.3.13 NEXTREC Specifier

The NEXTREC specifier has the form:

NEXTREC = nr

where:

nr

is an integer scalar memory reference.

Nr is assigned an integer value which is one more than the number of the last record read or written on the specified direct access file. If no records have been read or written, the value of nr is one. If the file is not connected for direct access, or if the position is indeterminate because of an error condition, nr is zero.

13.3.14 NUMBER Specifier

The NUMBER specifier has the form:

NUMBER = num

where:

num

is an integer scalar memory reference.

Num is assigned the number of the logical unit currently connected to the specified file. If there is no logical unit connected to the file, num is not defined.

13.3.15 OPENED Specifier

The OPENED specifier has the form:

OPENED = od

where:

od

is a logical scalar memory reference.

Od is assigned the value `.TRUE.` if the specified file is opened on a unit or if the specified unit is opened; it is assigned the value `.FALSE.` if the file or unit is not open.

13.3.16 ORGANIZATION Specifier

The ORGANIZATION specifier has the form:

ORGANIZATION= org

where:

org

is a character scalar memory reference.

Org is assigned the value `SEQUENTIAL` if the file is a sequential file, `RELATIVE` if the file is a relative file, and `INDEXED` if the file is an indexed file. If the processor is unable to determine the organization, org is assigned the value `UNKNOWN`.

13.3.17 RECL Specifier

The `RECL` specifier has the form:

`RECL = rcl`

where:

`rcl`

is an integer scalar memory reference.

If the file (or unit) is opened, `rcl` is the maximum record length allowed; if not opened, `rcl` is the longest record in the file. If a specified file does not exist, `rcl` is zero. `Rcl` is expressed in bytes if the file is opened for formatted I/O, and in longwords if the file is unformatted.

The `INQUIRE` statement reports the record length either in longwords (if a file has been previously opened for unformatted I/O) or in bytes (in all other circumstances).

13.3.18 RECORDTYPE Specifier

The `RECORDTYPE` specifier has the form:

`RECORDTYPE = rtype`

where:

`rtype`

is a character scalar memory reference.

`Rtype` is assigned the value `FIXED` if the file has fixed-length records, `VARIABLE` if the file has variable-length records, and `SEGMENTED` if the file is connected for unformatted sequential I/O using segmented records. If the file has stream-type records, `rtype` is assigned one of the following values: `STREAM` if the file's records are terminated with carriage-return and line-feed, `STREAM_CR` if they are terminated only with carriage-return, or `STREAM_LF` if they are terminated only with line-feed. If the processor cannot determine the record type, `rtype` is assigned the value `UNKNOWN`.

13.3.19 SEQUENTIAL Specifier

The `SEQUENTIAL` specifier has the form:

`SEQUENTIAL = seq`

where:

`seq`

is a character scalar memory reference.

Seq is assigned the value YES if SEQUENTIAL is an allowed access method for the specified file, NO if SEQUENTIAL is not an allowed access method, and UNKNOWN if the processor cannot determine whether SEQUENTIAL is an allowed access method.

13.3.20 UNFORMATTED Specifier

The UNFORMATTED specifier has the form:

```
UNFORMATTED = unf
```

where:

unf

is a character scalar memory reference.

Unf is assigned the value YES if unformatted is an allowed form for the file, NO if unformatted is not an allowed form for the file, and UNKNOWN if the processor is unable to determine whether unformatted is an allowed form for the file.

13.4 REWIND Statement

The REWIND statement repositions a sequential file currently open for sequential or append access to the beginning of the file. It has the forms:

```
REWIND ([UNIT=u],[ERR=s],[IOSTAT=ios])
```

```
REWIND u
```

where:

u

is a logical unit number.

s

is the label of the executable statement to which control is to be transferred if an error occurs.

ios

is an integer scalar memory reference that is assigned a positive integer if an error occurs, and zero if no error occurs.

The unit number must refer to a file on disk or magnetic tape.

For example:

```
REWIND 3
```

This statement repositions logical unit 3 to the beginning of the currently open file.

You must not issue a REWIND statement for a file that is open for direct or keyed access.

13.5 BACKSPACE Statement

The BACKSPACE statement repositions a sequential file currently open for sequential access to the beginning of the preceding record. When the next I/O statement for the unit is executed, this preceding record is available for processing.

The BACKSPACE statement has the forms:

```
BACKSPACE ([UNIT=]u[,ERR=s][,IOSTAT=ios])
```

```
BACKSPACE u
```

where:

u

is a logical unit number.

s

is the label of the executable statement to which control is to be transferred if an error occurs.

ios

is an integer scalar memory reference that is defined as a positive integer if an error occurs, and zero if no error occurs.

The unit number must refer to an open file on disk or magnetic tape. For example:

```
BACKSPACE 4
```

This statement repositions the open file on logical unit 4 to the beginning of the preceding record.

You must not issue a BACKSPACE statement for a file that is open for direct, keyed, or append access. Backspacing from record *n* is done by rewinding to the start of the file and then performing *n*-1 successive reads to reach the previous record. For direct, keyed, and append access, the current record count (*n*) is not available to the FORTRAN I/O system.

13.6 ENDFILE Statement

The ENDFILE statement writes an end-file record to the specified unit. It has the forms:

```
ENDFILE ([UNIT=]u[,ERR=s][,IOSTAT=ios])
```

```
ENDFILE u
```

where:

u

is a logical unit number.

s

is the label of the executable statement to which control is to be transferred if an error occurs.

ios

is an integer scalar memory reference that is defined as a positive integer if an error occurs, and zero if no error occurs.

An end-file record can be written only to sequential organization files that are accessed as formatted sequential or unformatted segmented sequential files.

For example:

```
ENDFILE 2
```

This statement writes an end-file record to logical unit 2.

You must not issue an ENDFILE statement for a file that is open for direct access.

End-file records should not be written in files that are read by programs written in a language other than FORTRAN because VAX RMS does not support the embedded end-file concept. An end-file record is a 1-byte record containing the hexadecimal code 1A (CTRL/Z).

13.7 DELETE Statement

The DELETE statement deletes records from relative and indexed files. It has the forms:

Indexed File Access

```
DELETE ([UNIT=]u[,ERR=s][,IOSTAT=ios])
```

Relative File Access

```
DELETE ([UNIT=]u,REC=r[,ERR=s][,IOSTAT=ios])
```

```
DELETE (u `r[,ERR=s][,IOSTAT=ios])
```

where:

u

is the number of the logical unit containing the record to be deleted.

r

is the positional number of the record to be deleted.

s

is the label of an executable statement to which control is to be transferred if an error condition occurs.

ios

is an integer scalar memory reference that is defined as a positive integer if an error occurs, and zero if no error occurs.

The form of the **DELETE** statement for use with indexed files is a current-record delete. This form of the statement deletes the current record, which is the last record to be accessed by a **READ** statement on the specified logical unit.

The forms of the **DELETE** statement for use with relative files are direct access deletes. These forms of the statement delete the record specified by the number *r*.

The **DELETE** statement logically removes the appropriate record from the specified file; that is, it locates the record and marks it as a deleted record. It then frees the position formerly occupied by the deleted record so that a new record can be written into that position.

After a direct access delete, any associated variable is set to the next record number.

The following examples demonstrate the use of the **DELETE** statement:

1. The fifth record in the file connected to logical unit 10 is deleted from the file in the following example.

```
DELETE (10,REC=5)
```

2. The current record is deleted from the file connected to logical unit 11 in the following example.

```
DELETE (11)
```

13.8 UNLOCK Statement

The UNLOCK statement unlocks a record in a relative or indexed file locked by a previous READ, without performing any other I/O operations. It has the forms:

```
UNLOCK ([unit=]u[,ERR=s][,IOSTAT=ios])
```

```
UNLOCK u
```

where:

u

is the number of a logical unit.

s

is the label of the executable statement to which control is to be transferred if an error occurs.

ios

is an integer scalar memory reference that is defined as a positive integer if an error occurs, and zero if no error occurs.

The UNLOCK statement frees a previously locked record on the specified logical unit. If no record is locked, the operation has no effect.

Chapter 14

Using Structures and Records

VAX FORTRAN structures and records allow you to easily group associated data together. Like arrays, records can contain one or more data elements. Unlike arrays however, the data elements in a record, called fields, can have different data types. Additionally, unlike arrays in which each element has a numeric index with which to uniquely identify it, each field of a record has a unique name.

This chapter provides an overview of how records can be used in VAX FORTRAN programs. Detailed information about the specifics of record use is split up among other chapters in this manual. Topics addressed in preceding chapters are as follows:

- The way to reference records and how records appear in memory (Section 6.2.5)
- The use of records in assignment statements (Section 7.4)
- The format of the RECORD statement (Section 8.13)
- The format of structure declaration blocks, which define the fields or groups of fields within a record (Section 8.15.1)

Aside from introducing terms relating to structured data items and their formation, the record construct has affected the terminology used to describe data items in general. It is important to understand the terminology changes relating to data items because they are in evidence throughout the manual. See Section 6.2.6 for a discussion of the terminology used to collectively refer to data items.

14.1 Structures

In VAX FORTRAN, structures are used to describe the form of records. You can think of structures as templates for records, defining the form and size of records.

Structures are defined with blocks of statements originating with a `STRUCTURE` statement and ending with an `END STRUCTURE` statement. Structure declaration blocks can contain the following statements:

- Statements that appear very much like data type declaration statements. These statements describe the fields contained within the structure.
- Statements that define substructures (nested structure declarations and `RECORD` statements) and mapped common areas (union declarations). These constructs are not discussed in this chapter; see Sections 8.15.2 and 8.15.3 for details.
- `PARAMETER` statements. A `PARAMETER` statement in a structure declaration block has its normal effect of assigning a symbolic name to a constant.

The name of a structure is specified in the `STRUCTURE` statement. The `RECORD` statement uses this name to identify the structure that is to be made into a record (or structured field) (see Section 14.2).

The following example defines the structure `DATE`. It contains three fields: `DAY`, `MONTH`, and `YEAR`. Note that the field `YEAR` is initialized with 1984. Any records defined to have the structure `DATE` will have their `YEAR` field initialized to 1984.

```
STRUCTURE /DATE/  
    LOGICAL*1 DAY, MONTH  
    INTEGER*2 YEAR /1984/  
END STRUCTURE
```

The following example defines the structure `PERSON`, which might be used to hold information about an individual. It contains the fields `NAME`, `SEX`, and `BIRTH__DATE`. Note that the fields `NAME` and `BIRTH__DATE` are themselves structured; that is, they are substructures within the structure `PERSON`. `NAME`'s structure declaration (unnamed) contains the fields `LAST__NAME`, `FIRST__NAME`, and `MIDDLE__INITIAL`. `BIRTH__DATE` has the structure of `DATE`, the structure defined in the preceding example.

```
STRUCTURE /PERSON/  
    STRUCTURE NAME  
        CHARACTER*20 LAST_NAME, FIRST_NAME  
        CHARACTER*1 MIDDLE_INITIAL  
    END STRUCTURE  
    LOGICAL*1 SEX  
    RECORD /DATE/ BIRTH__DATE  
END STRUCTURE
```

See Section 8.15 for detailed information about structure declarations and their syntactical elements.

14.2 Records

Records in FORTRAN are analogous to variables and arrays. Their “data type” is determined by the template, or structure, that is used to define them. The RECORD statement is used to define record scalars and arrays, in much the same way that type declaration statements are used. For example, the following RECORD statement, based on the structure PERSON shown in the preceding section, could be used.

```
RECORD /PERSON/ FATHER, MOTHER, CHILDREN(10)
```

The preceding statement creates twelve records with the structure PERSON. In all twelve records, all fields are initially undefined, with the exception of BIRTH__DATE.YEAR, which has been initialized to 1984 in the structure declaration DATE.

See Section 6.2.5 for information about how to reference records and fields and about how they appear in memory. See Section 8.13 for detailed information about the syntax of RECORD statements.

14.3 Uses of Records

When you have several data arrays, each containing a different, though related, type of information, the use of records allows you to use the same index to refer to each array.

As an example, consider a FORTRAN program which maintains and manipulates a symbol table. The symbol table consists of three arrays: the first contains the symbol names, the second contains the symbol values, and the third contains a flag signaling whether the symbol is defined. As an example, the declaration in FORTRAN-77 could be as follows:

```
PARAMETER (MAXSYM=1000)
CHARACTER*16 SYMBOL_NAME(MAXSYM)
INTEGER*4     SYMBOL_VALUE(MAXSYM)
LOGICAL*1     SYMBOL_FLAG(MAXSYM)
```

Note that each array is declared separately and that, although the data items are related, they are declared (and later manipulated) separately. For example, to read or write such related information from or to a file, you must specify each piece individually, such as in the following WRITE statement:

```
WRITE (10) SYMBOL_NAME(I), SYMBOL_VALUE(I), SYMBOL_FLAG(I)
```

With structures and records, however, the definition allows you to group the related information together, and refer to them as a whole in many cases. Instead, then, the symbol table declaration could appear as follows:

```
STRUCTURE /SYM/
  CHARACTER*16  NAME
  INTEGER*4     VALUE
  LOGICAL*1     FLAG
END STRUCTURE
. . .
RECORD /SYM/ SYMBOL(MAXSYM)
```


With these definitions, there is only one array, the record array SYMBOL. Each element of SYMBOL has the form, or structure, of SYM. This means that each element of SYMBOL consists of the three fields NAME, VALUE, and FLAG. Note that the related information about an individual symbol—its name, value, and defined-flag—are now one element of a record array. As a result, you can refer to a symbol table (that is, SYMBOL instead of individual arrays such as SYMBOL__NAME), a single symbol I (for example, SYMBOL(I) instead of SYMBOL__NAME(I)), or any of the fields in symbol I (for example, SYMBOL(I).NAME). Thus, the previous WRITE statement would be changed to:

```
WRITE (10) SYMBOL(I)
```

This statement is equivalent to:

```
WRITE (10) SYMBOL(I).NAME, SYMBOL(I).VALUE, SYMBOL(I).FLAG
```

In some cases, such as with arguments of system service calls, FORTRAN programs had to use COMMON blocks to pass structured information to subroutines (see the *VAX FORTRAN User's Guide* for information on data structure arguments). Routines such as these expect the address of either a list, control block, or vector, and the COMMON statement constructs these arguments, with no empty spaces between adjacent items, in order of declaration. The resulting COMMON block(s) can be used as records but do not have the flexibility of records.

For example, a call to the SYS\$GETJPI system service requires the address of a sequence of items consisting of two words followed by two longwords. With records, this call can be achieved with the following code:

```
STRUCTURE /GETJPI_ITEM/
    INTEGER*2  W_LEN, W_CODE
    INTEGER*4  L_ADDR, L_LENADDR/0/
END STRUCTURE
RECORD /GETJPI_ITEM/GETJPIARG(5)
...
GETJPIARG(4).W_LEN = 4
GETJPIARG(4).W_CODE = JPI$_CPUTIM
GETJPIARG(4).L_ADDR = %LOC(LCL_VALUES(4))
...
CALL SYS$GETJPI(,,,GETJPIARG,,,)
```

As this example illustrates, the primary advantage to using records is that they enable you to group related data together in one conceptual whole.

Chapter 15

Using Indexed Files

Traditionally, sequential and direct access have been the only file access modes available to FORTRAN programs. To overcome some of the limitations of these access modes, VAX FORTRAN supports a third access mode, called keyed access (see Section 11.2.3.4). Keyed access allows you to retrieve records, at random or in sequence, based on key fields that are established when you create a file with indexed organization.

You can access files with indexed organization using either sequential or keyed access, or a combination of both.

1. Keyed access retrieves records randomly based on the particular key fields and key values that you specify.
2. Sequential access retrieves records in an ascending sequence based on the values within the particular key field that you specify.

The combination of keyed and sequential access is commonly referred to as the Indexed Sequential Access Method (ISAM). Once you have read a record by means of an indexed read request, you can then use a sequential read request to retrieve records with ascending key field values, beginning with the key field value in the record retrieved by the initial read request.

Indexed organization is especially suitable for maintaining complex files in which you want to select records based on one of several criteria. For example, a mail-order firm could use an indexed organization file to store its customer list. Key fields could be a unique customer order number, the customer's zip code, and the item ordered. Reading sequentially based on the zip code key would enable you to produce a mailing list sorted by zip code. A similar operation based on customer order number or item number key would enable you to list the records in customer order number or item number sequence.

The remainder of this chapter provides information of the following major topics:

- Creating an indexed file (Section 15.1)
- Writing records to an indexed file (Section 15.3)
- Reading records from an indexed file (Section 15.5)
- Deleting records from an indexed file (Section 15.4)
- Updating records in an indexed file (Section 15.4)

Information is also provided about the effects of read and write operations on positioning your program to records within an indexed file (Section 15.6) and about how to build logic into your programs to handle exception conditions that commonly occur (Section 15.7).

15.1 Creating an Indexed File

You can create a file with an indexed organization by using either the FORTRAN OPEN statement or the RMS EDIT/FDL utility.

- Use the OPEN statement to specify the file options supported by FORTRAN.
- Use the EDIT/FDL utility to select features not directly supported by FORTRAN.

Any indexed file created with EDIT/FDL can be accessed by FORTRAN I/O statements.

When you create an indexed file, you define certain fields within each record as key fields. One of these key fields, called the *primary key*, is identified as key number zero and must be present in every record. Additional keys, called *alternate keys*, can also be defined; they are numbered from 1 through a maximum of 254. An indexed file can have as many as 255 key fields defined. In practice, however, few applications require more than 3 or 4 key fields.

The data types used for key fields must be either INTEGER*2, INTEGER*4, or CHARACTER.

In designing an indexed file, you must decide the byte positions of the key fields. For example, in creating an indexed file for use by a mail-order firm, you might define a file record to consist of the following fields:

```
STRUCTURE /FILE_REC_STRUCT/
  INTEGER*4 ORDER_NUMBER      ! Positions 1:4, Key 0
  CHARACTER*20 NAME           ! Positions 5:24
  CHARACTER*20 ADDRESS        ! Positions 25:44
  CHARACTER*19 CITY           ! Positions 45:63
  CHARACTER*2 STATE           ! Positions 64:65
  CHARACTER*9 ZIP_CODE        ! Positions 66:74, Key 1
  INTEGER*2 ITEM_NUMBER       ! Positions 75:76, Key 2
END STRUCTURE
.
.
.
RECORD /FILE_REC_STRUCT/ FILE_REC
```

Given this record definition, you could use the following OPEN statement to create an indexed file:

```
OPEN (UNIT=10, FILE='CUSTOMERS.DAT', STATUS='NEW',
1     ORGANIZATION='INDEXED', ACCESS='KEYED',
2     RECORDTYPE='VARIABLE', FORM='UNFORMATTED',
3     RECL=19,
4     KEY=(1:4:INTEGER, 66:74:CHARACTER, 75:76:INTEGER),
5     IOSTAT=IOS, ERR=9999)
```

This OPEN statement establishes the attributes of the file, including the definition of a primary key and two alternate keys. Note that the definitions of the integer keys do not explicitly state INTEGER*4 and INTEGER*2. The data type sizes are determined by the number of character positions allotted to the key fields, which in this case are 4 and 2 character positions, respectively.

If you specify the KEY keyword when opening an existing file, the key specification that you give must match that of the file.

FORTTRAN uses RMS default key attributes when creating an indexed file. These defaults are as follows:

- The values in primary key fields cannot be changed when a record is rewritten and cannot have duplicates.
- The values in alternate key fields can be changed and can have duplicates.

You can use the EDIT/FDL utility or a USEROPEN routine to override these defaults and to specify other values not supported by VAX FORTRAN, such as null key field values, null key names, and key data types other than integer and character.

Refer to *VAX FORTRAN User's Guide* for information on the use of the USEROPEN keyword in OPEN statements. The *VAX Record Management Services Reference Manual* has more information on indexed file options.

Use of the EDIT/FDL utility is explained in detail in the *VAX/VMS File Definition Language Facility Reference Manual*.

15.2 Writing Indexed Files

You can write records to an indexed file with either formatted or unformatted indexed WRITE statements. Each write operation inserts a new record into the file and updates the key index(es) so that the new record can be retrieved in a sequential order based on the values in the respective key fields.

For example, you could add a new record to the file for the mail-order firm (see Section 15.1) with the following statement:

```
WRITE (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC
```

The following two sections describe considerations that relate to write operations: (1) the effects of writing records with duplicate values in key fields and (2) the method by which you can prevent an alternate key field in a record from being indexed during a write operation.

15.2.1 Duplicate Values in Key Fields

It is possible to write two or more records with the same value in a single key field. The attributes specified for the file when it was created determine whether this duplication is allowed. By default, FORTRAN creates files that allow duplicate alternate key field values

and prohibit duplicate primary key field values (see Section 15.1). If duplicate key field values are present in a file, the records with equal values are retrieved on a first-in/first-out basis.

For example, assume that five records are written to an indexed file in this order (for clarity, only key fields are shown):

ORDER__NUMBER	ZIP__CODE	ITEM__NUMBER
1023	70856	375
942	02163	2736
903	14853	375
1348	44901	1047
1263	33032	690

If the file is later opened and read sequentially by primary key (ORDER__NUMBER), the order in which the records are retrieved is not affected by the duplicated value (375) in the ITEM__NUMBER key field. In this case, the records would be retrieved in the following order:

ORDER__NUMBER	ZIP__CODE	ITEM__NUMBER
903	14853	375
942	02163	2736
1023	70856	375
1263	33032	690
1348	44901	1047

However, if the read operation is based on the second alternate key (ITEM__NUMBER), the order in which the records are retrieved is affected by the duplicate key field value. In this case, the records would be retrieved in the following order:

ORDER__NUMBER	ZIP__CODE	ITEM__NUMBER
1023	70856	375
903	14853	375
1263	33032	690
1348	44901	1047
942	02163	2736

Notice that the records containing the same key field value (375) are retrieved in the order in which they were written to the file.

15.2.2 Preventing the Indexing of Alternate Key Fields

When writing to an indexed file that contains variable-length records, you can prevent entries from being added to the key index(es) for any alternate key field(s). This is done by omitting the names of the alternate key field(s) from the WRITE statement. The omitted alternate key field(s) must be at the end of the record; another key field cannot be specified after the omitted key field.

For example, assume that the last record (ORDER__NUMBER 1263) in the mail-order example is written with the following statement:

```
WRITE (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC.ORDER_NUMBER,  
1 FILE_REC.NAME, FILE_REC.ADDRESS, FILE_REC.CITY,  
1 FILE_REC.STATE, FILE_REC.ZIP_CODE
```

Because the field name FILE__REC.ITEM__NUMBER is omitted from the WRITE statement, an entry for that key field is not created in the index. As a result, an attempt to read the file using the alternate key ITEM__NUMBER would not retrieve the last record and would produce the following listing:

ORDER__NUMBER	ZIP__CODE	ITEM__NUMBER
1023	70856	375
903	14853	375
1348	44901	1047
942	02163	2736

You can omit only trailing alternate keys from a record; the primary key must always be present.

15.3 Reading Indexed Files

You can read records in an indexed file with either sequential or indexed READ statements (formatted or unformatted) under the keyed mode of access. By specifying ACCESS='KEYED' in the OPEN statement, you enable both sequential and keyed access to the indexed file.

Indexed READ statements position the file pointers (see Section 15.6) at a particular record, determined by the key field value, the key-of-reference, and the match criterion. Once you retrieve a particular record by an indexed READ statement, you can then use sequential access READ statements to retrieve records with increasing key field values.

The form of the external record's key field must match the form of the value you specify in the KEY keyword. Thus, if the key field contains character data, you should specify the KEY keyword value as a CHARACTER data type. If the key field contains binary data, then the KEY keyword value should be of INTEGER data type.

Note that if you write a record to an indexed file with formatted I/O, the data type is converted from its internal representation to an external representation. As a result, the key value must be specified in the external form when you read the data back with an indexed read. Otherwise, a match will occur when you do not expect it.

The following FORTRAN program segment prints the order number and zip code of each record where the first five characters of the zip code are greater than or equal to '10000' but less than '50000':

```
C   Read first record with ZIP_CODE key greater than or
C   equal to '10000'.

      READ (UNIT=10,KEYGE='10000',KEYID=1,IOSTAT=IOS,ERR=9999)
      1   FILE_REC

C   While the zip code previously read is within range, print
C   the order number and zip code, then read the next record.

      DO WHILE (FILE_REC,ZIP_CODE .LT. '50000')
      PRINT *, 'Order number', FILE_REC,ORDER_NUMBER, 'has zip code',
      1   FILE_REC,ZIP_CODE
      READ (UNIT=10,IOSTAT=IOS,END=200,ERR=9999)
      1   FILE_REC

C   END= branch will be taken if there are no more records
C   in the file.

      END DO
200   CONTINUE
```

The error branch on the keyed READ in this example is taken if no record is found with a zip code greater than or equal to '10000'; an attempt to access a nonexistent record is an error. If the sequential READ has accessed all records in the file, however, an end-of-file status occurs, just as with other file organizations.

If you wish to detect a failure of the keyed READ, you can examine the I/O status variable, IOS, for the appropriate error number (see Table 18-1 for a list of the returned error codes).

15.4 Updating Records

The REWRITE statement updates existing records in an indexed file. You cannot replace an existing record simply by writing it again; a WRITE statement would attempt to add a new record.

An update operation is accomplished in two steps. First, you must read the record in order to make it the current record. Next, you execute the REWRITE statement. For example, to update the record containing ORDER_NUMBER 903 (see prior examples) so that the NAME field becomes 'Theodore Zinck', you might use the following FORTRAN code segment:

```
READ (UNIT=10,KEY=903,KEYID=0,IOSTAT=IOS,ERR=9999) FILE_REC
FILE_REC,NAME = 'Theodore Zinck'
REWRITE (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC
```

When you rewrite a record, key fields may change. The attributes specified for the file when it was created determine whether this type of change is permitted.

15.5 Deleting Records

To delete records from an indexed file, you use the `DELETE` statement. The `DELETE` and `REWRITE` statements are similar; a record must first be locked by a `READ` statement before it can be operated on.

The following FORTRAN code segment deletes the second record in the file with `ITEM_NUMBER` 375 (refer to previous examples):

```
READ (UNIT=10,KEY=375,KEYID=2,IOSTAT=IOS,ERR=9999)
READ (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC
IF (FILE_REC,ITEM_NUMBER .EQ. 375) THEN
    DELETE (UNIT=10, IOSTAT=IOS, ERR=9999)
ELSE
    PRINT *, 'There is no second record.'
END IF
```

Deletion removes a record from all defined indexes in the file.

15.6 Current Record and Next Record Pointers

The RMS file system maintains two pointers into an open indexed file: the “next record” pointer and the “current record” pointer.

- The next record pointer indicates the record to be retrieved by a sequential read. When you open an indexed file, the next record pointer indicates the record with the lowest primary key field value. Subsequent sequential read operations cause the next record pointer to be the one with the next higher value in the same key field. In case of duplicate key field values, records are retrieved in the order in which they were written.
- The current record pointer indicates the record most recently retrieved by a `READ` operation; it is the record that is locked from access by other programs sharing the file. The current record is the one operated on by the `REWRITE` statement (see Section 11.6) and the `DELETE` statement (see Section 13.7). The current record is undefined until a read operation is performed on the file. Any file operation other than a read causes the current record pointer to become undefined. Also, an error results if a rewrite or delete operation is performed when the current record pointer is undefined.

15.7 Exception Conditions

You can expect to encounter certain exception conditions when using indexed files. The two most common of these conditions involve valid attempts to read locked records and invalid attempts to create duplicate keys. Provisions for handling both of these situations should be included in a well-written program.

When an indexed file is shared by several users, any read operation may result in a "specified record locked" error. One way to recover from this error condition is to ask if the user would like to reattempt the read. If the user's response is positive, then the program can go back to the READ statement. For example:

```
INCLUDE '($FORIOSDEF)'  
  
100 READ (UNIT=10,IOSTAT=IOS) DATA  
  
IF (IOS .EQ. FOR$IOS_SPERECLOC) THEN  
    TYPE *, 'That record is locked. Press RETURN'  
    TYPE *, 'to try again, or CONTROL_Z to discontinue'  
    READ (UNIT=*,FMT=*,END=900)  
    GO TO 100  
ELSE IF (IOS .NE. 0) THEN  
    CALL ERROR (IOS)  
END IF
```

You should avoid simply looping back to the READ statement without first providing some type of delay (caused by a request to try again, or to discontinue, as in this example). If your program reads a record but does not intend to modify the record, you should place an UNLOCK statement immediately after the READ statement. This technique reduces the time that a record is locked and permits other programs to access the record.

The second exception condition, creation of duplicate keys, occurs when your program tries to create a record with a key field value that is already in use. When duplicate key field values are not desirable, you might have your program prompt for a new key field value whenever an attempt is made to create a duplicate. For example:

```
INCLUDE '($FORIOSDEF)'  
  
200 WRITE (UNIT=10,IOSTAT=IOS) KEY_VAL, DATA  
  
IF (IOS .EQ. FOR$IOS_INCKEYCHG) THEN  
    TYPE *, 'This key field value already exists. Please enter'  
    TYPE *, 'a different key field value, or press CONTROL_Z'  
    TYPE *, 'to discontinue this operation.'  
    READ (UNIT=*,FMT=300,END=999) KEY_VAL  
    GO TO 200  
ELSE IF (IOS .NE. 0) THEN  
    CALL ERROR (IOS)  
END IF
```

Chapter 16

Using Character Data

VAX FORTRAN's character data type allows you to easily manipulate alphanumeric data. You can use character data in the form of character variables, arrays, constants, and expressions. A character operator (//) is available to form character strings by concatenating character elements.

16.1 Character Substrings

You can select certain segments (substrings) from a character variable or character array element by specifying the name of the variable or array element, followed by delimiter values indicating the leftmost and/or rightmost characters in the substring. For example, if the character variable NAME contained the string

```
ROBERT△WILLIAM△BOB△JACKSON
```

and you wished to extract the substring BOB, you would specify the following:

```
NAME(16:18)
```

If you omit the first value, you are indicating that the first character of the substring is the first character in the variable. For example, if you specify

```
NAME(:18)
```

the resulting substring is

```
ROBERT△WILLIAM△BOB
```

If you omit the second value, you are specifying the rightmost character to be the last character in the variable. For example:

```
NAME(16:)
```

encompasses

```
BOB△JACKSON
```

16.2 Building Character Strings

It is sometimes useful to create strings from two or more separate strings. This is done by means of the concatenation operator, the double slash (//). For example, you might wish to create a variable called NAME, consisting of the values of the following variables:

```
FIRSTNAME  
MIDDLENAME  
NICKNAME  
LASTNAME
```

To do so, define each as a character variable of a specified length. For example:

```
CHARACTER*42 NAME  
CHARACTER*12 FIRSTNAME,MIDDLENAME,LASTNAME  
CHARACTER*6 NICKNAME
```

Concatenation is accomplished as follows:

```
NAME = FIRSTNAME//MIDDLENAME//NICKNAME//LASTNAME
```

Thus, if the variables contained the values

```
FIRSTNAME = 'ROBERT'  
MIDDLENAME = 'WILLIAM'  
NICKNAME = 'BOB'  
LASTNAME = 'JACKSON'
```

which are stored individually as

```
ROBERT△△△△△  
WILLIAM△△△△△  
BOB△△△  
JACKSON△△△△△
```

then, when concatenated and stored in NAME, they become the string:

```
ROBERT△△△△△WILLIAM△△△△△BOB△△△JACKSON△△△△△
```

Applying the substring extraction facility described in Section 16.1, you can get the stored nickname by specifying

```
NAME(25:30)
```

which picks up the 6-character substring BOB△△△ (including trailing blanks) in variable NAME.

16.3 Character Constants

Character constants are strings of characters enclosed in apostrophes. You can assign a character value to a character variable in much the same way you would assign a numeric value to a real or integer variable. For example:

```
XYZ = 'ABC'
```

As a result of this statement, the characters ABC are stored in location XYZ. Note that if XYZ's length is less than three bytes, the character string is truncated on the right. Thus, if you specify

```
CHARACTER*2 XYZ
```

```
XYZ = 'ABC'
```

the result is AB. If, on the other hand, the variable is longer than the constant, it is padded on the right with blanks. For example:

```
CHARACTER*6 XYZ
```

```
XYZ = 'ABC'
```

results in having

```
ABC△△△
```

stored in XYZ. The previous contents of XYZ are overwritten. Thus, if the previous contents of XYZ were CBSNBC, the result would still be ABC△△△.

You can give character constants symbolic names by using the PARAMETER statement. For example:

```
CHARACTER*(*) TITLE  
PARAMETER (TITLE = 'THE METAMORPHOSIS')
```

The PARAMETER statement in the preceding example assigns the symbolic name TITLE to the character constant THE METAMORPHOSIS.

You can use the symbolic name TITLE anywhere a character constant is allowed.

To include an apostrophe as part of the constant, specify two consecutive apostrophes. For example:

```
CHARACTER*(*) TITLE  
PARAMETER (TITLE = 'FINNEGAN''S WAKE')
```

results in the character constant FINNEGAN'S WAKE.

The value assigned to a character parameter can be any compile-time constant character expression. Note in particular that the CHAR intrinsic function (see Section 16.7.1) with a constant argument is a compile-time constant expression; therefore, you can assign non-printing characters to parameter constants. For example:

```
CHARACTER*(*) CRLF  
PARAMETER (CRLF=CHAR(13)//CHAR(10))
```

16.4 Declaring Character Data

To declare variables or arrays as character type, use the CHARACTER type declaration statement, as shown in the following example:

```
CHARACTER*10 TEAM(12), PLAYER
```

This statement defines a 12-element character array (TEAM), each element of which is 10 bytes long, and a character variable (PLAYER), which is also 10 bytes long.

You can specify different lengths for variables in a CHARACTER statement by including a length value for specific variables. For example:

```
CHARACTER*6 NAME, AGE*2, DEPT
```

In this example, NAME and DEPT are defined as 6-byte variables, while AGE is defined as a 2-byte variable.

16.5 Initializing Character Variables

Use the DATA statement to preset the value of a character variable. For example:

```
CHARACTER*10 NAME, TEAM(5)
DATA NAME/' ', TEAM/'SMITH','JONES',
1      'DOE','BROWN','GREEN'/
```

Note that NAME contains 10 blanks, while each array element in TEAM contains the corresponding character value, right-padded with blanks.

To initialize an array so that each of its elements contains the same value, use a DATA statement of the following type:

```
CHARACTER*5 TEAM(10)
DATA TEAM/10*'WHITE'/
```

The result is a 10-element array in which each element contains WHITE.

You can also initialize character variables within the character declaration, as shown in the following example:

```
CHARACTER*10 METALS(3)/'LEAD','IRON','GOLD'/
```

16.6 Passed-Length Character Arguments

In writing subprograms that manipulate character data, you can get the subprogram to accept actual character arguments of any length by specifying the length of the dummy argument as passed-length. To indicate a passed-length dummy argument, use an asterisk (*) as follows:

```
SUBROUTINE REVERSE(S)
CHARACTER*(*) S
  *
  *
  *
```

The passed-length notation indicates that the length of the actual argument is used when processing the dummy argument string. This length can change from one invocation of the subprogram to the next. For example:

```
CHARACTER A*20, B*53
      .
      .
      .
CALL REVERSE(A)
CALL REVERSE(B)
```

In the first call to REVERSE, the length of S is 20; in the second call, its length is 53.

You can use the CHARACTER*(*) notation to define the length of parameter character constants. The actual length is then the length of the character constant that is assigned to the parameter name in a PARAMETER statement.

The FORTRAN function LEN can be used to determine the actual length of the string (see Section 16.7.4).

16.7 Character Library Functions

VAX FORTRAN supports the following character functions:

- CHAR
- ICHAR
- INDEX
- LEN
- LGE, LGT, LLE, LLT

The following sections describe these functions.

16.7.1 CHAR Function

The CHAR function returns a 1-byte character value equivalent to the integer ASCII code value passed as its argument. It has the form:

```
CHAR(i)
```

where:

i

is an integer expression equivalent to an ASCII code.

16.7.2 ICHAR Function

The ICHAR function returns an integer ASCII code equivalent to the character expression passed as its argument. It has the form:

ICHAR(c)

where:

c
is a character expression. If **c** is longer than one byte, the ASCII code equivalent to the first byte is returned and the remaining bytes are ignored.

16.7.3 INDEX Function

The INDEX function is used to determine the starting position of a substring. It has the form:

INDEX(c1,c2)

where:

c1
is a character expression that specifies the string to be searched for a match with the value of **c2**.

c2
is a character expression representing the substring for which a match is desired.

If INDEX finds an instance of the specified substring (**c2**), it returns an integer value corresponding to the starting location in the string (**c1**). For example, if the substring sought is CAT, and the string that is searched contains DOGCATFISHCAT, the return value of INDEX is 4.

If INDEX cannot find the specified substring, it returns the value 0.

If there are multiple occurrences of the substring, INDEX locates the first (leftmost) one. Use of the INDEX function is illustrated in Figures 16-1 and 16-2.

16.7.4 LEN Function

The LEN function returns an integer value that indicates the length of a character expression. It has the form:

LEN(c)

where:

c
is a character expression.

16.7.5 LGE, LGT, LLE, LLT Functions

The lexical comparison functions LGE, LGT, LLE, and LLT are defined by the FORTRAN-77 standard to make comparisons between two character expressions using the ASCII collating sequence. The result is the logical value `.TRUE.` if the lexical relation is true, and `.FALSE.` if the lexical relation is not true. The functions have the forms:

```
LGE(c1,c2)
LGT(c1,c2)
LLE(c1,c2)
LLT(c1,c2)
```

where:

c1,c2

are character expressions.

You may wish to include these functions in FORTRAN programs that can be used on computers that do not use the ASCII character set. In VAX FORTRAN, the lexical comparison functions are equivalent to the `.GE.`, `.GT.`, `.LE.`, `.LT.` relational operators. For example:

```
IF (LLE(string1, string2)) GO TO 100
```

is equivalent to

```
IF (string1 .LE. string2) GO TO 100
```

16.8 Character Data Examples

An example of character data usage is shown in Figures 16-1 and 16-2. The example in Figure 16-1 is a program that manipulates the letters of the alphabet. The results are shown in Figure 16-2.

16.9 Character I/O

The character data type simplifies the transmission of alphanumeric data. You can read and write character strings of any length from 1 to 65535 characters. For example:

```
CHARACTER*24 TITLE
      .
      .
      .
READ (12,100) TITLE
100 FORMAT (A)
```


These statements cause 24 characters read from logical unit 12 to be stored in the 24-byte character variable `TITLE`. Compare this with the code necessary if you used Hollerith data stored in numeric variables or arrays:

```
INTEGER*4 TITLE(6)
      .
      .
      .
READ (12,100) TITLE
100 FORMAT (6A4)
```

Note that you must divide the data into lengths suitable for real or (in this case) integer data and specify I/O and `FORMAT` statements to match. In this example, a one-dimensional array comprising six 4-byte elements is filled with 24 characters from logical unit 12.

```
CHARACTER C, ALPHABET*26

DATA ALPHABET/'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/

WRITE (6,90)
90  FORMAT (' CHARACTER EXAMPLE PROGRAM OUTPUT'/)

DO I=1,26
  WRITE (6,*) ALPHABET
  ALPHABET = ALPHABET(2:)//ALPHABET(1:1)
END DO

CALL REVERSE(ALPHABET)
WRITE (6,*) ALPHABET

CALL REVERSE(ALPHABET(1:13))
WRITE (6,*) ALPHABET

CALL FIND_SUBSTRINGS('UVW', ALPHABET)
CALL FIND_SUBSTRINGS('A', 'DAJHDHAJDAHDJA4E CEUEBCUEIAWSAWQLQ')

WRITE (6,*) 'END OF CHARACTER EXAMPLE PROGRAM'
END
```

```

SUBROUTINE REVERSE(S)
CHARACTER T, S*(*)

J = LEN(S)
DO I=1,J/2
    T = S(I:I)
    S(I:I) = S(J:J)
    S(J:J) = T
    J = J - 1
END DO
END

SUBROUTINE FIND_SUBSTRINGS(SUB,S)
CHARACTER*(*) SUB, S
CHARACTER*132 MARKS

I = 1
MARKS = ' '

10 J = INDEX(S(I:),SUB)
IF (J .NE. 0) THEN
    I = I + (J-1)
    MARKS(I:I) = ' '
    I = I+1
    IF (I .LE. LEN(S)) GO TO 10
END IF

WRITE (6,91) S, MARKS
91 FORMAT (2(/1X,A))
END

```

Figure 16-1: Character Data Program Example

```
CHARACTER EXAMPLE PROGRAM OUTPUT

ABCDEFGHIJKLMNPOQRSTUVWXYZ
BCDEFGHIJKLMNPOQRSTUVWXYZA
CDEFGHIJKLMNPOQRSTUVWXYZAB
DEFGHIJKLMNPOQRSTUVWXYZABC
EFGHIJKLMNPOQRSTUVWXYZABCD
FGHIJKLMNPOQRSTUVWXYZABCDE
GHIJKLMNPOQRSTUVWXYZABCDEF
HIJKLMNPOQRSTUVWXYZABCDEFG
IJKLMNPOQRSTUVWXYZABCDEFGH
JKLMNOPQRSTUVWXYZABCDEFGHIJ
KLMNOPQRSTUVWXYZABCDEFGHIJ
LMNOPQRSTUVWXYZABCDEFGHIJK
MNOPQRSTUVWXYZABCDEFGHIJKL
NOPQRSTUVWXYZABCDEFGHIJKLM
OPQRSTUVWXYZABCDEFGHIJKLMN
PQRSTUVWXYZABCDEFGHIJKLMNO
QRSTUVWXYZABCDEFGHIJKLMNOP
STUVWXYZABCDEFGHIJKLMNOPQR
TUVWXYZABCDEFGHIJKLMNOPQRS
UVWXYZABCDEFGHIJKLMNOPQRST
VWXYZABCDEFGHIJKLMNOPQRSTU
WXYZABCDEFGHIJKLMNOPQRSTU
XYZABCDEFGHIJKLMNOPQRSTU
YZABCDEFGHIJKLMNOPQRSTU
ZABCDEFGHIJKLMNOPQRSTU
ZYXWVUTSRQPONMLKJIHGFEDCBA
NOPQRSTUVWXYZMLKJIHGFEDCBA

#

DAJHDHAJDAHDJA4E CEUEBCUEIAWSAWQLQ
# # # # # #
END OF CHARACTER EXAMPLE PROGRAM
```

ZK-793-82

Figure 16-2: Output Generated by Example Program

Chapter 17

Debugging VAX FORTRAN Programs

This chapter describes how to use VAX DEBUG (or, simply, the debugger).

The *VAX/VMS Symbolic Debugger Reference Manual* describes the debugger in detail. Refer to that manual or access on-line HELP information from a debugging session for detailed descriptions of the syntax of commands discussed in this chapter.

Topics covered by the individual sections are as follows:

- Section 17.1 provides a general description of the debugger.
- Section 17.2 explains how to compile and link a program to be analyzed using the debugger.
- Section 17.3 explains how to invoke and terminate the debugger.
- Section 17.4 describes debugger help, command entry options, and debugger command procedures and their use. It also provides a summary of debugger commands and their syntax.
- Section 17.5 describes how to start, stop, and control a program while you are running it under the control of the debugger.
- Section 17.6 describes what you must know about how the debugger treats the symbolic names in your programs and explains how to designate program locations.
- Section 17.7 explains how to examine variables and program locations and how to modify their contents while you are debugging a program.
- Section 17.8 explains how to control debugger screen displays.
- Section 17.9 presents simple examples of debugging FORTRAN programs.

17.1 Debugging Overview

Debugging, the process of locating and correcting errors, is one of the most difficult stages in program development. You need to debug when one or more of the following situations occur:

- Compile-time errors are signaled.
- Run-time errors are signaled.
- You determine, based on receiving incorrect output during a program's execution, that a logic error exists.

The VAX FORTRAN compiler and run-time system display error and warning messages when errors occur. You can use this information to determine where the error exists in your program and then to correct it.

You must detect errors that produce incorrect output yourself. To help you find such errors, VAX/VMS provides a special program: VAX DEBUG. It enables you to control the execution of your program so that you can monitor specific locations, change the contents of locations, check the sequence of program control, and otherwise locate and correct errors as they occur. After you track down the mistakes, you can edit your source program and repeat the compile-link-execute sequence with the corrected version.

The VAX Symbolic Debugger has many helpful features, among which are the following:

- It is interactive. You control your program and interact with the debugger from your terminal.
- It understands FORTRAN variables and their data types. Thus, when you want to look at or change the value of a variable, the debugger will display the value in a manner appropriate to the data type or convert your ASCII text input to the data type of the variable.
- It understands many other programming languages as well, such as PASCAL and PL/I. Thus, if your program consists of routines written in different languages, you can change from one language to another during the course of a single debugging session.

17.2 Preparing a Program for Debugging—Compiling and Linking

To execute a VAX FORTRAN program with the debugger, you should first compile the program with the /DEBUG and /NOOPTIMIZE qualifiers on the FORTRAN command and then link the program with the /DEBUG qualifier on the LINK command.

You should specify the /NOOPTIMIZE qualifier on the FORTRAN command because several of the optimizations performed by the compiler can cause the program and the debugger commands to behave in unexpected ways. Note: The VAX FORTRAN compiler,

by default, optimizes the object code in order to make it run faster during execution. See Chapter 1 in *VAX FORTRAN User's Guide* for detailed descriptions of the various optimizations performed by the compiler and how these affect debugging.

For a description of the effects of specifying the /DEBUG qualifier on FORTRAN, LINK, and RUN commands, see Sections 3.2.3.4 and 4.3.

When you no longer need to use the debugger on your program, recompile the program with the default qualifiers of /NODEBUG and /OPTIMIZE.

17.3 Invoking and Terminating the Debugger

You can invoke, interrupt, and terminate the debugger in a variety of ways. This flexibility can be an important tool during the debugging process. The various options are described in the subsections that follow.

17.3.1 Invoking the Debugger with the RUN Command

If your program has been compiled and linked with the /DEBUG qualifier, the RUN command passes initial control to the debugger rather than commencing program execution.

The following example shows how the debugger identifies itself:

```
$ RUN CIRCLE
```

```
VAX DEBUG Version 4.1-2
```

```
%DEBUG-I-INITIAL, language is FORTRAN, module set to 'CIRCLE'  
DBG>
```

The module name displayed in the debugger's message is the name of the main program unit, which is associated with the executable image's entry point. In the example, the debugger message indicates that the name of the main program unit is CIRCLE.

The DBG> prompt indicates that the debugger is ready to process your commands. You respond to the prompt with any of the commands recognized by the debugger.

If your program was not linked with the /DEBUG qualifier, you can still have initial control passed to the debugger by specifying /DEBUG on the RUN command; however, your debugging options will be limited (see Section 4.3).

To avoid automatically invoking the debugger when you are running a program compiled and linked with the /DEBUG qualifier, specify the /NODEBUG qualifier on your RUN command. As described in the next section, even though you specify /NODEBUG, you can still access the debugger during the execution of your program.

17.3.2 Invoking the Debugger During Program Execution

You can interrupt an executing program at any time by entering <CTRL/Y>. You can then invoke the debugger by entering the DEBUG command—even if you specified /NODEBUG on the RUN command. This method of interrupting an executing program can be useful, for example, if you think that the program is looping or if you see erroneous output.

17.3.3 Suspending the Debugger to Issue DCL Commands

At any point in a debugging session, you can use the debugger's SPAWN command to create a subprocess that allows you to use DCL commands without terminating the debugging session.

If you specify the SPAWN command with a DCL command as a parameter, the spawned subprocess executes the command and immediately returns control to the debugger. The following example executes MAIL in the middle of a debugging session:

```
DBG> SPAWN MAIL
MAIL> READ
.
.
MAIL> EXIT
DBG>
```

If you specify the SPAWN command without a DCL command as a parameter, you can enter any number of DCL commands before returning to your debugging session. To resume your debugging session, specify the LOGOUT command as shown in the following example:

```
DBG> SPAWN
$ MAIL
.
.
MAIL> EXIT
$ LOGOUT
  Process USER_1 logged out at 28-JUL-1984 09:59:12:45
%DEBUG-I-RETURNED, control returned to process USER
DBG>
```

The debugger also has an ATTACH command. The debugger's ATTACH command works the same way as the ATTACH command in DCL.

17.3.4 Interrupting the Debugger

Entering <CTRL/Y> terminates any debugger command that is executing and puts you at DCL command level. You can then return to the debugger with the DCL commands CONTINUE or DEBUG. The CONTINUE command continues the debugging session as though no interruption had occurred. The DEBUG command returns control to debugger command level, allowing you to issue new debugger commands (useful for aborting an infinite loop or a long operation associated with a debugger command).

17.3.5 Terminating a Debugger Session

The following message indicates that your program has executed normally:

```
%DEBUG-I-EXITSTATUS, is ' %SYSTEM-S-NORMAL, normal successful completion '  
DBG>
```

To terminate the debugging session, type <CTRL/Z> or EXIT:

```
DBG> EXIT  
$
```

The dollar sign prompt indicates that you are at DCL command level.

17.4 The Debugger Environment

The debugger provides you with a subsystem for entering commands—much like MAIL and EDT, for example. Many features that are available from DCL are also available in this subsystem. This includes keypad definitions, HELP capability, command files, and so forth. In some cases, DEBUG's syntax is different from DCL's syntax. This section explains the DEBUG sub-environment.

The following topics are addressed in this section:

- Debugger HELP information
- Options relating to how you enter commands and command procedures
- Debugger commands (presented in an alphabetical list that shows all of the optional qualifiers and parameters)

17.4.1 Using Debugger HELP

To display the list of debugger commands on which information is available, type HELP. To display information about a particular command, type HELP plus the command name. For example, to display information about the use of the qualifier /ALL with the SET MODULE command, specify:

```
DBG> HELP SET MODULE/ALL
```


17.4.2 Entering Commands

You can enter debugging commands in two ways: by typing out the commands on your terminals' normal keyboard or by using keypad keys.

17.4.2.1 Normal Keyboard Entry

To enter commands on the main keyboard, you type in the command and then press <RET>. You can enter more than one command on a line by separating the commands with semicolons (;). In addition, you can continue a command on a new line by ending the line with a hyphen (-), and the debugger will then prompt for the remainder of the command line with an underscore (_).

Debugger commands, like DCL commands, can be abbreviated to unique characters. For example, the command CANCEL EXCEPTION BREAK could be entered as CAN EX BR. In addition, you can abbreviate a debugger command by using the DEFINE command with its /COMMAND qualifier to equate the command to a shorter symbolic name. The following example creates the symbol CEB to abbreviate the command CANCEL EXCEPTION BREAK:

```
DBG> DEFINE/COMMAND CEB = 'CANCEL EXCEPTION BREAK'
```

To make your symbol definitions available at each debugging session, include them in a debug initialization command file that is executed at the start of all debugging sessions (see Section 17.4.5).

17.4.2.2 Keypad Entry

You can also enter debugging commands by means of the keypad. Figure 17-1 shows the default keypad definitions. You can redefine each of these keys with the DEFINE/KEY command (see Section 17.4.3). The DCL and MAIL keypad entry rules work in a similar way.

Each key can represent up to three debug commands. The first command is selected by pressing the key by itself; the second by pressing the key in combination with the PF1 (GOLD) key; the third by pressing the key in combination with the PF4 (BLUE) key.

Each key also has a symbolic name associated with it. You can use this symbolic name to define a key or key combination to represent debug commands of your own choosing. In Figure 17-1, the symbolic names are shown in at the top of the key.

To enter a debug command in keypad mode, you first press the desired key or key combination. Some commands, such as the GO and SET MODE commands, require no additional parameters. When these keys are pressed, the command is executed immediately. Other commands require you to enter a parameter. When those keys are pressed, you type the parameter on the main keypad. The command is executed when you press either the <RET> or <ENTER> key.

	PF1	PF2	PF3	PF4
Default	GOLD	HELP	SET MODE SCREEN	BLUE
GOLD	GOLD	HELP	SET MODE NOSCR	BLUE
BLUE	GOLD	HELP	DISP/GENERATE	BLUE
	KP7	KP8	KP9	MINUS
Default		SCROLL/UP	DISPLAY next	DISP next AT FS
GOLD		SCROLL/TOP		
BLUE		SCROLL/UP *		DISP SRC, OUT
	KP4	KP5	KP6	COMMA
Default	SCROLL/LEFT	EX/SOU .0\%PC	SCROLL/RIGHT	GO
GOLD	SCROLL/LEFT:132	SHOW CALLS		
BLUE	SCROLL/LEFT *	SHOW CALLS 3	SCROLL/RIGHT *	
	KP1	KP2	KP3	ENTER
Default	EXAMINE	SCROLL/DOWN	SEL/SCROLL next	
GOLD		SCROLL/BOTTOM	SEL/OUTPUT next	
BLUE		SCROLL/DOWN *	SEL/SOURCE next	
	KP0		PERIOD	
Default	STEP		Reset	
GOLD	STEP/INTO		Reset	
BLUE	STEP/OVER		Reset	

ZK-1732-84

Figure 17-1: Default Keypad Definitions—VT100

17.4.3 User-Defined Keypad Command Keys

The following commands are used in examining and changing key definitions:

- DEFINE/KEY** Creates a key definition, which associates a debug command of your own choosing with a key or key combination.
- DELETE/KEY** Deletes a key definition.
- SHOW KEY** Displays the definition for a specified key.

The debugger DEFINE/KEY command (similar to the DCL DEFINE/KEY command) allows you to assign a debugger command to a keypad key. For example, to define the keypad key 7 to enter and execute the SET MODULE/ALL command, specify:

```
DBG> DEFINE/KEY/TERMINATE KP7 'SET MODULE/ALL'
```

You must be in keypad mode to define, use, display, or delete a keypad key. To display the current definition of a keypad key, specify:

```
DBG> SHOW KEY key
```

To delete a key's definition, specify:

```
DBG> DELETE/KEY key
```

You can put key definitions in a debugger initialization file so that the key is available whenever the initialization file is executed (see Section 17.4.5).

Key Name	Key Designation
PF1	LK201, VT100, VT52 Red
PF2	LK201, VT100, VT52 Blue
PF3	LK201, VT100, VT52 Black
PF4	LK201, VT100
KP0, KP1, ..., KP9	Keypad 0 - 9
PERIOD	Keypad period
COMMA	Keypad comma
MINUS	Keypad minus
ENTER	Keypad enter
E1	LK201 Find
E2	LK201 Insert Here
E3	LK201 Remove
E4	LK201 Select
E5	LK201 Prev Screen
E6	LK201 Next Screen
HELP	LK201 Help
DO	LK201 Do
F6, F7, ..., F20	LK201 Function keys

17.4.4 Using Debugger Command Procedures

Like DCL, the debugger has a capability for executing a sequence of commands contained in a file. The syntax is the same as for DCL:

```
DBG> @filename
```

You can execute a command procedure interactively, from within a DO command sequence, or from within another command procedure. Command procedures are especially useful when you regularly perform a number of standard set-up debugger commands; see Section 17.4.5 for information about initialization files.

To display the commands in a command procedure (or DO command sequence) as they execute, specify the VERIFY keyword of the SET OUTPUT command:

```
DBG> SET OUTPUT VERIFY
```

If you want the debugger to accept input commands automatically from a file whenever you invoke the debugger, assign the logical name DBG\$INPUT to point to the file before invoking the debugger. For example:

```
$ TYPE EXIT.COM  
EXIT  
$ ASSIGN EXIT.COM DBG$INPUT  
$ RUN/DEBUG PROG  
DBG> EXIT  
$
```

In the preceding example, the debugger accepts its input from the file EXIT.COM, thus causing it to exit immediately.

17.4.5 Initializing a Debugging Session

The logical name DBG\$INIT can be used to specify a command file that is to be executed whenever you start a debugging session. This capability is analogous to the LOGIN.COM file that is always executed at the start of a terminal session.

The following commands are examples of commands that are commonly used in initialization files:

```
SET OUTPUT LOG,VERIFY  
SET LOG filename  
SET MODULE/ALL  
SET MODE SCREEN  
SET STEP SILENT
```

Note that in addition to establishing a “generic” initialization command procedure that would be invoked whenever you access the debugger, you may also want to establish special initialization command procedures for use with individual programs that require repeated debugging because of their complexity or size.

17.4.6 Recording Debug Sessions in Log Files

A debugger log file maintains a history of a debugging session. Each debugger command and display that occurs during a debugging session is stored in the log file.

The DBG> prompt is not recorded and the displays are commented out with exclamation points in order to allow the use of log files as command procedures. Thus, if a lengthy debugging session is interrupted for some reason, you can execute the log file as you would any other debugger command procedure and it will restore your debugging session to the point at which it was previously terminated.

To create a log file, specify the following debugger commands:

- **SET LOG filename**—specifies the name of the log file.
- **SET OUTPUT LOG**—directs the debugger to begin to send output to a log file (as well as to the terminal).

For example:

```
DBG> SET LOG HIST.LOG
DBG> SET OUTPUT LOG
```

The default file specification of a log file is **DEBUG.LOG**.

To use a log file as a command procedure, invoke it as follows:

```
DBG> SET OUTPUT VERIFY
DBG> @HIST.LOG
%DEBUG-I-VERIFYICF, entering indirect command file 'HIST.LOG'
  SET BREAK /CALL/TEMPORARY
  .
  .
  .
%DEBUG-I-VERIFYICF, exiting indirect command file 'HIST.LOG'
DBG>
```

17.4.7 Debugger Command Syntax and Summary

Debugger commands have the format:

```
cmd [keyword] [/qualifier] [param ...] !comment
```

cmd

A command verb (for example, **CANCEL**, **SET**, **SHOW**) that indicates the general function to be performed.

keyword

Gives the specific function to be performed by the command verb (for example, **CANCEL MODULE**, **SET SCOPE**, **SHOW LANGUAGE**).

/qualifier

Modifies the effect of the command. Qualifiers change the defaults that the debugger uses to process commands. For example, when you deposit a value, the debugger uses decimal radix by default. You can override the default by specifying either **/HEXADECIMAL** or **/OCTAL**.

param

Qualifies the function in some way, such as specifying a range of locations to be acted upon by the command.

comment

Any text message. The debugger ignores all text after the exclamation mark.

Separate the command, keyword, and operand fields by one or more spaces. Multiple commands can be specified on one line by using a semicolon (;) as a command separator. The debugger prompts for commands with the symbol DBG>.

Table 17-1 summarizes the debugger commands.

Table 17-1: Summary of Debug Commands

@filespec

ALLOCATE n-bytes

ATTACH process-name

CALL routine [(arg [,arg ...])]

CANCEL ALL

CANCEL BREAK $\left[\begin{array}{l} /BRANCH \\ /CALL \\ /EXCEPTION \\ /INSTRUCTION [=opcode] \\ /LINE \\ /MODIFY \end{array} \right] \left\{ \begin{array}{l} /ALL \\ breakpoint [,breakpt ...] \end{array} \right\}$

CANCEL DISPLAY $\left\{ \begin{array}{l} /ALL \\ display-name \end{array} \right\}$

CANCEL EXCEPTION BREAK

CANCEL MODE

CANCEL MODULE $\left\{ \begin{array}{l} /ALL \\ module-name \end{array} \right\}$

CANCEL RADIX [/OVERRIDE]

CANCEL SCOPE

CANCEL SOURCE [/MODULE=module-name]

CANCEL TRACE $\left[\begin{array}{l} /BRANCH \\ /CALL \\ /EXCEPTION \\ /INSTRUCTION [=opcode] \\ /LINE \\ /MODIFY \end{array} \right] \left\{ \begin{array}{l} /ALL \\ tracept [,tracept ...] \end{array} \right\}$

Table 17-1 (Cont.): Summary of Debug Commands

CANCEL TYPE/OVERRIDE

CANCEL WATCH { /ALL
watchpt [,watchpt ...] }

CANCEL WINDOW { /ALL
window-name [,window-name ...] }

DECLARE name [:kind] [,name [:kind]]

DEFINE [/ADDRESS
/VALUE
/COMMAND
/GLOBAL] symbol = expression [,symbol = expression ...]

DEFINE/KEY [/[(NO)]ECHO
/[(NO)]IF_STATE
/[(NO)]LOCK_STATE
/[(NO)]LOG
/[(NO)]SET_STATE
/[(NO)]TERMINATE] keyname expression

DELETE/KEY [/ALL
/[(NO)]LOG
/[(NO)]STATE] keyname

DEPOSIT [/ASCII:n
/ASCIC
/ASCIW
/ASCIZ
/BYTE
/D_FLOAT
/FLOAT
/G_FLOAT
/H_FLOAT
/INSTRUCTION
/LONG
/OCTAWORD
/QUADWORD
/WORD] { addr-expressn = expression }

DISPLAY [/CLEAR
/GENERATE
/HIDE
/MARK_CHANGE
/REFRESH
/REMOVE
/SIZE:n] [display-name [AT window-name] [kind]] ,...

Table 17-1 (Cont.): Summary of Debug Commands

EVALUATE	[/ADDRESS /BINARY /CONDITION__VALUE /DECIMAL /HEXADECIMAL /OCTAL]	expression [,expression ...]
EXAMINE	[/ASCII:n /ASCIC /ASCID /ASCIW /ASCIZ /BINARY /BYTE /CONDITION__VALUE /D__FLOAT /DECIMAL /FLOAT /G__FLOAT /H__FLOAT /HEXADECIMAL /INSTRUCTION /LONG /OCTAL /OCTAWORD /QUADWORD /SOURCE /[NO]SYMBOL /WORD]	addr-expressn [,addr-expressn ...]

EXIT

EXITLOOP [n-level]

FOR name = expression TO expression [BY expression] DO (debug-cmds)

GO

HELP [topic]

IF language-expression THEN (debug-cmds) [ELSE (debug-cmds)]

REPEAT language-expression DO (debug-cmds)

SAVE old-display AS new-display

SCROLL	[/BOTTOM /DOWN /LEFT /RIGHT /TOP /UP]	[display-name]
--------	--	----------------

Table 17-1 (Cont.): Summary of Debug Commands

SEARCH	[/ALL /NEXT /IDENTIFIER /STRING]	[range] string
SELECT	[/OUTPUT /SCROLL /SOURCE]	display-name
SET BREAK	[/AFTER:n /BRANCH /CALL /EXCEPTION /INSTRUCTION [=opcode] /LINE /MODIFY /[NO]SOURCE /RETURN /[NO]SILENT /TEMPORARY]	addr-expressn[,addr-expressn...] [WHEN (condition-expressn)] [DO (debug-cmds)]
SET DISPLAY	[/MARK_CHANGE /REMOVE /SIZE:n]	[display-name [AT window] [kind]] , ...
SET EXCEPTION BREAK		
SET LANGUAGE language		
SET LOG filespec		
SET MARGIN	[right-margin left-margin:right-margin left-margin: :right-margin]	
SET MAX_SOURCE_FILES n-files		
SET MODE mode [,mode ...]		
SET MODULE [/ALLOCATE]	{ /ALL module-name [,module-name ...] }	
SET OUTPUT	[[NO]LOG [NO]SCREEN_LOG [NO]TERMINAL [NO]VERIFY]	

Table 17-1 (Cont.): Summary of Debug Commands

SET RADIX	[/INPUT /OUTPUT /OVERRIDE]	[BINARY DECIMAL DEFAULT HEXADECIMAL OCTAL]		
SET SCOPE	[/MODULE]	location [,location ...]		
SET SEARCH	[ALL NEXT IDENTIFIER STRING]			
SET SOURCE	[/MODULE=module-name]	filespec		
SET STEP	[BRANCH CALL EXCEPTION INSTRUCTION[=opcode] INTO LINE OVER RETURN [NO]SILENT [NO]SOURCE [NO]SYSTEM]			
SET TRACE	[/AFTER:n /BRANCH /CALL /EXCEPTION /INSTRUCTION [=opcode] /LINE /MODIFY /[NO]SOURCE /RETURN /[NO]SILENT /TEMPORARY]	addr-express [,addr-express ...]		
SET TYPE [/OVERRIDE]	{ ASCII ASCID ASCII:n ASCIZ	{ BYTE D_FLOAT DATE_TIME FLOAT	{ G_FLOAT H_FLOAT INSTRUCTION LONG	{ OCTAWORD PACKED:n QUADWORD WORD

Table 17-1 (Cont.): Summary of Debug Commands

SET TERMINAL/WIDTH:n

SET WATCH

/AFTER:n /SILENT /SOURCE /TEMPORARY	addr-express ,... [WHEN (condition)]
--	--------------------------------------

SET WINDOW name AT (line, line)

SHOW BREAK

SHOW CALLS [n-calls]

SHOW DISPLAY

SHOW KEY

/BRIEF /DIRECTORY /STATE=statname	{ <table border="0" style="display: inline-table; vertical-align: middle;"> <tr> <td style="padding: 0 5px;">/ALL</td> <td style="padding: 0 5px;">key-name</td> <td style="padding: 0 5px;">[,key-name ...]</td> </tr> </table> }	/ALL	key-name	[,key-name ...]
/ALL	key-name	[,key-name ...]		

SHOW LANGUAGE

SHOW LOG

SHOW MARGINS

SHOW MAX_SOURCE_FILES

SHOW MODE

SHOW MODULE

SHOW OUTPUT

SHOW SCOPE

SHOW SEARCH

SHOW SOURCE

SHOW STEP

SHOW SYMBOL

/ADDRESS /DIRECT /TYPE	symbol ,... [IN scope ,...]
------------------------------	-----------------------------

SHOW TRACE

SHOW TYPE [/OVERRIDE]

SHOW WATCH

SHOW WINDOW

SPAWN /NOWAIT dcl-command

STEP

/BRANCH /CALL /EXCEPTION /INSTRUCTION [=opcode] /INTO /LINE /OVER /RETURN /SILENT /[NO]SOURCE /[NO]SYSTEM	[n-units]
---	-------------

Table 17-1 (Cont.): Summary of Debug Commands

SYMBOLIZE addr-expression
TYPE [module\]line[:line] ,...

UNDEFINE $\left[\begin{array}{l} /ALL \\ /GLOBAL \\ /KEY \end{array} \right]$ symbol

WHILE language-expression DO (debug-cmds)

17.5 Controlling Program Execution

After you have invoked the debugger (see Section 17.3), you must decide how you wish to control the execution of your program. The debugger provides several commands, with a wide range of options, for you to select from.

Using these commands, you can suspend execution of your program at points that are interesting to you, and you can then examine variables at those points.

The following sections, Sections 17.5.1 and 17.5.2, contain information on the following topics concerning execution control:

- Starting program execution (GO, STEP, and CALL commands)
- Suspending or tracing program execution (SET BREAK, SET TRACE, and SET WATCH commands)

17.5.1 Starting Program Execution

The command with which you start program execution determines when the debugger regains control and prompts you for other debugger commands that may aid in isolating the error in your program.

The GO, STEP, and CALL commands execute varying numbers of source lines before returning control to the debugger. The SHOW CALLS command displays the current hierarchy of routine calls.

Table 17-1 contains complete specifications for each of these debugger commands.

17.5.1.1 GO Command

The GO command starts execution at the current line, continuing it either to the conclusion of the program (as in the following example), to an error, or to the next breakpoint or watchpoint (see Section 17.5.2):

```
DBG> GO
%DEBUG-I-EXITSTATUS,is '%SYSTEM-S-NORMAL,normal successful completion'
DBG>
```

An optional parameter of the GO command allows you to specify an address at which to start program execution. However, this alters the normal flow of the program and is likely to produce meaningless results.

17.5.1.2 STEP Command

By default, the STEP command executes one source statement. To execute more than one statement, specify the number of statements to be executed as a parameter of the STEP command. The following command executes the next two source statements:

```
DBG> STEP 2
stepped to MAIN\%LINE 5
      5:          CALL SUB2(J)
```

If you are debugging in screen mode (see Section 17.8), the information displayed by the STEP command is extraneous since the SRC display shows the source code and your current position. Use the SET STEP SILENT command to prevent the STEP command from displaying any text.

If the STEP command encounters a subprogram invocation, the following keywords determine whether the subprogram executes as a single step:

- OVER—indicates that the debugger executes subprograms as a single STEP command.
- INTO—indicates that the debugger steps through subprograms line by line.

To display the current default keywords in effect for the STEP command, enter the SHOW STEP command. The following SHOW STEP command displays the default keywords for the STEP command:

```
DBG> SHOW STEP
STEP TYPE:  NOSYSTEM, SOURCE, OVER ROUTINE CALLS, BY LINE
```

By default, the debugger executes all subprograms as a single step. To set the STEP characteristics so that the debugger steps through user subprograms, but not system subprograms, specify:

```
DBG> SET STEP NOSYSTEM, SOURCE, INTO, LINE
```

The LINE keyword in the previous example (same as BY LINE in the SHOW STEP display) indicates that a STEP command executes source line by source line. Alternatively, you can use the following keywords to specify that a STEP command executes all lines up to an exception or a particular type of machine code instruction (the BRANCH, CALL, and INSTRUCTION keywords are useful only if you are familiar with machine code):

- BRANCH—step to the next machine code branch instruction.
- CALL—step to the next machine code call instruction.
- EXCEPTION—step to the next exception (error).

- **INSTRUCTION**—step to the next machine code instruction.
- **LINE**—step to the next source code statement.
- **RETURN**—step to the end of the currently executing subprogram.

To use one of the previously listed keywords to affect a single **STEP** command, name the keyword as a qualifier of the **STEP** command (for example, **STEP/INSTRUCTION**). To use one of the previously listed keywords as the default for the **STEP** command, use the keyword as a parameter of the **SET STEP** command (for example, **SET STEP INSTRUCTION**).

17.5.1.3 CALL Command

The **CALL** command invokes a subprogram (passing it specified arguments), executes the subprogram, and displays the function value returned (none for a subroutine).

Typically, you use the **CALL** command to invoke a subprogram that you have written in order to display data structures or other information required for debugging.

When passing arguments to a subprogram invoked by the **CALL** command, you can use the following keywords to specify the passing mechanism:

- **%DESCR**—pass by descriptor. Unlike FORTRAN, you must use **%DESCR** explicitly when specifying **CHARACTER** type arguments.
- **%REF**—pass by reference. Unlike FORTRAN, you must use **%REF** explicitly when specifying expression arguments that do not refer to specific locations. For example, to pass **A*B**, you must specify **%REF (A*B)**.
- **%VAL**—pass by value.

For example, the subprogram **INC_DUMP** is a subroutine that requires two arguments, both passed by reference. To invoke **INC_DUMP**, specify:

```
DBG> CALL INC_DUMP (PERSONS_HOUSE, ADULTS_HOUSE)
value returned is 0
```

17.5.1.4 SHOW CALLS Command

The **SHOW CALLS** command produces a traceback of calls and is particularly useful when you have returned to the debugger following a **<CTRL/Y>** interrupt or a program exception. For each call frame (beginning with the most recent call), the debugger displays one line of information, including the name of the routine, the name of the module containing the routine, and the line number of the call. For example:

```
DBG> SHOW CALLS
  module name      routine name      line      rel PC      abs PC
  SUB1             SUB1              4         0000000C    0000042C
*MAIN             MAIN              4         00000011    00000411
```

The name of the routine and the name of the module containing the routine are always the same for FORTRAN subroutines. (Note: This is not the case with other languages.)

The value of the program counter (PC) in the calling subroutine at the time that control passed to the called subroutine is also displayed. The is expressed both as a virtual address relative to the virtual address of the subroutine's name and as an absolute address.

The asterisk (*) indicates which modules are set (SET MODULE command).

17.5.2 Suspending or Tracing Program Execution

You can suspend program execution at specified locations in your program by setting breakpoints and watchpoints with the SET BREAK and SET WATCH commands. In addition, you can follow program execution without suspending execution by setting tracepoints with the SET TRACE command.

An optional WHEN clause allows you to conditionally activate a breakpoint, tracepoint, or watchpoint. An optional DO clause allows you to specify one or more debugger commands to be executed when a breakpoint, tracepoint, or watchpoint is activated. SHOW and CANCEL commands display and cancel the breakpoints, tracepoints, and watchpoints that you have set.

NOTE

You cannot set breakpoints, tracepoints, and watchpoints at the same location: the most recently issued command overrides any other breakpoint, tracepoint, or watchpoint at that location.

Table 17-1 contains complete specifications for each of these debugger commands.

17.5.2.1 Breakpoints and Tracepoints

You can set a breakpoint at a particular program location, on an exception, or on a particular type of instruction. When your program encounters a breakpoint, the debugger suspends program execution, displays the address of the breakpoint and the source line at that address, executes the DO command sequence (if specified), and prompts for a command (unless the DO command sequence causes an alternative action).

A tracepoint is exactly like a breakpoint, except that instead of prompting for a command, the debugger executes an implicit GO command to continue program execution.

The options associated with the SET BREAK and SET WATCH commands are described in the following list:

- To set a breakpoint or tracepoint at a particular program location, specify that location as the parameter of the SET BREAK or SET TRACE command. The following example sets a breakpoint that causes the debugger to suspend execution just before line 3 in subroutine SUB1:

```
DBG> SET MODULE/ALL
DBG> SET BREAK SUB1\%LINE 3
DBG> GO
break at SUB1\%LINE 3
      3:          M = M +1
```

A breakpoint usually suspends execution at the first byte of the specified location so that the instruction beginning at that location does not execute. However, if you set a breakpoint at a subroutine, the breakpoint is actually set at the memory address two bytes greater than the address of the subroutine name itself (the entry point), thereby causing the subroutine to be called before the debugger takes control and issues the message: "routine break at routine NAME."

- To set a breakpoint or tracepoint on an exception or on a particular type of instruction, use the following qualifiers of the SET BREAK and SET TRACE commands (note that these qualifiers are the same as those used on the STEP and SET STEP commands):
 - /BRANCH—break on the next machine code branch instruction.
 - /CALL—break on the next machine code call instruction.
 - /EXCEPTION—break on the next exception (error). The debugger reports the exception and the line at which it occurred, after which you can execute or inhibit a user-declared condition handler. You cannot use the STEP command to step into a condition handler, but you can set a breakpoint or tracepoint within the handler. (The SET BREAK/EXCEPTION command is the same as the SET EXCEPTION BREAK command.)
 - /INSTRUCTION—break on the next machine code instruction.
 - /LINE—break on the next source code statement.
 - /RETURN—break on the end of the currently executing subprogram.

- To have a breakpoint or tracepoint execute a list of commands when it is activated, use a DO command sequence.

```
DBG> SET BREAK SUB1 DO (EXAMINE M)
```

The preceding command specifies that the EXAMINE M command is to be executed each time that the breakpoint at subroutine SUB1 is encountered.

- To activate a tracepoint or breakpoint exactly once, specify the /TEMPORARY qualifier. For example:

```
DBG> SET BREAK/TEMPORARY %LINE 8
DBG> GO
break at CIRCLE\%LINE 8
      8:          AREA = PI*RADIUS**2
```

- To activate a tracepoint or breakpoint after a certain number of iterations, specify the /AFTER qualifier. For example, to activate a tracepoint on the second execution of line 3 in program CIRCLE, specify:

```
DBG> SET TRACE/AFTER:2 %LINE 3 DO (IF I .EQ. 2 THEN (GO) ELSE (EXAMINE I))
DBG> GO
trace at CIRCLE\%LINE 3
      3: 4          TYPE 5
trace at CIRCLE\%LINE 3
      3: 4          TYPE 5
CIRCLE\I:          3
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
```

The tracepoint is activated at each successive execution of that line (unless you specify /TEMPORARY).

- To conditionally execute a breakpoint or tracepoint, use the optional WHEN clause. Each time the debugger encounters the breakpoint or tracepoint, it evaluates the expression in the WHEN clause: if the expression is true, the breakpoint or tracepoint is activated; if it is false, the breakpoint or tracepoint is ignored. For example, in the following command, the DO command sequence executes only if the expression RADIUS .LT. 10 is true:

```
DBG> SET TRACE %LINE 8 WHEN (RADIUS .LT. 10) DO (DEPOSIT RADIUS = 0)
```

The SHOW BREAK and SHOW TRACE commands display the breakpoints and tracepoints currently set in the program. You can cancel any of the currently set breakpoints and tracepoints by means of the CANCEL BREAK or CANCEL TRACE commands. For example:

```
DBG> CANCEL BREAK %LINE 9
```

This command cancels the breakpoint at line 9. To cancel all breakpoints, enter the following:

```
DBG> CANCEL BREAK/ALL
```

The commands to cancel tracepoints have the same syntax as the preceding examples.

17.5.2.2 Watchpoints

Specify a watchpoint to display a particular program location each time that the contents of that location are modified. When your program modifies a value in the specified location, the debugger performs the following operations:

- Suspends program execution.
- Displays the address of the location, the old and new values of the location, and the source line that modified the location.
- Executes the DO command sequence (if specified).
- Prompts for a command (unless the DO command sequence causes an alternative action).

To set a watchpoint on a location, specify that location as the parameter of the SET WATCH command. The following example sets a watchpoint on the location I in the program unit MAIN and starts execution:

```
DBG> SET MODULE/ALL
DBG> SET WATCH I
DBG> GO
watch of MAIN\I at SUB1\%LINE 3
  3:      M = M + 1
      old value: 1
      new value: 2
break at SUB1\%LINE 4
  4:      RETURN
DBG>
```

The /AFTER and /TEMPORARY qualifiers and the DO and WHEN commands can be used with the SET WATCH command in the same way that they are used with the SET BREAK and SET TRACE commands. See Section 17.5.2.1 for examples of the /AFTER and /TEMPORARY qualifiers, as well as the DO and WHEN clauses.

The SHOW and CANCEL commands work the same way with watchpoints as they do with breakpoints and tracepoints (see Section 17.5.2.1).

17.5.3 Displaying Source Lines

Debugger commands allow you to display source lines under a variety of circumstances. The STEP/SOURCE and SET MODE SCREEN commands display source lines as they execute; the TYPE and SEARCH commands display source lines independently. All commands that display source lines require the following:

1. The source file must have been compiled with the /DEBUG and /NOOPTIMIZE qualifiers.

2. The source file must reside in the same directory in which it was compiled. If not, you must use the SET SOURCE command to establish the file's new directory. For example:

```
DBG> SET SOURCE [directory1][,directory2]...
```

Note that by specifying a directory list, you can tell the debugger which directories it should search (in the order specified) to find the files.

3. When necessary, you must specify the appropriate scope or, alternatively, the appropriate pathname.

The STEP/SOURCE command displays the currently executing source lines; see Section 17.5.1.2 for information about the STEP command.

The SET MODE SCREEN command generates the SRC display by default; SRC shows the currently executing source line, the lines preceding it, and the lines following it; see Section 17.8 for information on debugger screen displays.

The TYPE and SEARCH commands display source lines independent of their execution:

- TYPE—displays a specified range of source lines. For example, to display lines 1 through 5 of module SUB1, type:

```
DBG> SET MODULE/ALL
DBG> TYPE SUB1\1:5
module SUB1
  1:          SUBROUTINE SUB1(M)
  2:          INTEGER M,N/5/, L/31/
  3:          M = M + 1
  4:          RETURN
  5:          END
```

In SCREEN mode, the source lines appear in the source display (SRC, by default); in NOSCREEN mode, the source lines appear with other debugger output.

- SEARCH—displays the source lines containing the specified string. For example, to display all lines containing the string M in the module named SUB1, specify:

```
DBG> SEARCH/ALL SUB1 M
module SUB1
  1:          SUBROUTINE SUB1(M)
  2:          INTEGER M,N/5/, L/31/
  3:          M = M + 1
```

The SEARCH display appears with other debugger output.

17.5.4 Using the Debugger's Logical Control Commands

You can control the execution of debugger commands by using the following logical control commands:

- **IF THEN ELSE**—a conditional construct that executes a THEN clause of one or more debugger commands if a specified logical expression is true. If the expression is false, the command either terminates or executes an optional ELSE clause of one or more debugger commands. The format of an IF construct is:

```
IF expression THEN (command[;...]) [ELSE(command[;...])]
```

In the following example, if the expression in the IF clause is false, the ELSE clause executes:

```
DBG> IF I .GT. 100 THEN (EXAMINE X) ELSE (GO)
```

- **WHILE**—an iterative construct that executes a DO command sequence while a specified logical expression is true; if the expression is false, the command terminates. The format of a WHILE construct is:

```
WHILE expression DO(command[;...])
```

The following example causes the debugger to step line by line while I is less than 5:

```
DBG> WHILE I .LT. 5 DO(STEP/LINE)
```

- **FOR**—an iterative construct that executes a DO command sequence through a range of values. FOR control = init TO term [BY inc] DO(command[;...])

The control variable is initialized to the value of init and compared to term. If control is less than term, the commands in the DO command sequence execute and control is incremented by one (or the value of inc). If control is less than term, the command terminates. For example, the following FOR command displays the value of three variables 3 times:

```
DBG> FOR I = 1 TO 3 DO(EXAMINE RADIUS,AREA)
```

Logical control constructs are especially useful in command procedures and DO command sequences (see Section 17.4.4 for information about using debugger command procedures).

17.6 Using Symbolic Names and Accessing Program Locations

During a debugging session you can reference the following items:

1. Locally defined symbolic names, such as names of variables
2. Globally defined symbolic names, such as names of subroutines
3. Virtual memory locations, such as an address returned by a system-defined routine

4. Debugger symbols for VAX registers:

<code>%R0 - %R11</code>	General registers 0 through 11
<code>%AP</code>	Argument pointer
<code>%FP</code>	Frame pointer
<code>%SP</code>	Stack pointer
<code>%PC</code>	Program counter
<code>%PSL</code>	Processor status longword

5. User-defined debugger symbols, as described in Section 17.6.4

Access to symbols described by items 1 and 2 in the preceding list depends on whether you specify `/DEBUG` when you compile and link your program. Also, note that some of these symbols may disappear as a result of optimizations performed at compile time. Thus, in addition to `/DEBUG`, you should specify `/NOOPTIMIZE` when you compile a program that you intend to run using the debugger (see Section 17.2).

Much of the information in this section is needed only when you are debugging multiunit programs and you must exercise control over the debugger symbol table (see Section 17.6.1) or resolve “not unique” debugger messages (see Section 17.6.3). Information in this section is not necessary if you are debugging a single unit program and are just examining and manipulating values in variables. This section describes how to access “lower level” constructs (such as absolute addresses, registers, or instructions) that are seldom needed.

17.6.1 Making Symbolic Names Accessible—`SET MODULE`

The `MODULE` commands (`SET`, `SHOW`, and `CANCEL`) enable you to control the contents of the debugger’s symbol table when the program you want to debug consists of multiple program units. These commands perform the following functions:

- `SET MODULE` places the symbols defined in the specified program unit or units in the symbol table. By default, the debugger initializes the symbol table to include all global symbols and the local symbols of the first program unit specified in the `LINK` command.
- `SHOW MODULE` displays the names of all program units whose symbols are potentially available. “Yes” means the symbols for that module are in the symbol table; “No” means they are not.
- `CANCEL MODULE` removes the specified program unit’s symbols from the symbol table.

For example, to make the symbols in the program unit `SUB2` available while operating in `SUB1` (see sample program `MAIN` in Section 17.9.2) specify:

```
DBG> SET MODULE SUB2
```

If your program is not too large, it is useful to make the symbols of all modules available at the outset of the debugging session. (Performance will suffer noticeably if your program is too large for all of its modules to be set.) To set all modules, specify the `/ALL` qualifier with the `SET MODULE` command:

```
DBG> SET MODULE/ALL
```

Depending upon the size of your program, most or all of the symbols in its modules will be set. To display the names of modules whose symbols are available, use the debugger command `SHOW MODULE`. The following example demonstrates the effect of the `SET MODULE/ALL` command. At the outset of a debugging session on a program named `MAIN`, only the symbols of the first linked module (the main program unit named `MAIN`) are available:

```
DBG> SHOW MODULE
module name                symbols    language    size

MAIN                       yes       FORTRAN     660
SUB1                       no        FORTRAN     440
SUB2                       no        FORTRAN     468
SHARE#DEBUG                no        Image       0
SHARE#DBGSSISHR           no        Image       0
SHARE#LIBRTL               no        Image       0
SHARE#FORRTL               no        Image       0
SHARE#LBRSHR               no        Image       0
SHARE#MTHRTL               no        Image       0
SHARE#PLIRTL               no        Image       0
SHARE#SCRSHR               no        Image       0
SHARE#SMGSHR               no        Image       0

total modules: 12.          remaining size: 54140.
```

Entering the `SET MODULE/ALL` command makes the symbols in all of the modules available:

```
DBG> SET MODULE/ALL
DBG> SHOW MODULE
module name                symbols    language    size

MAIN                       yes       FORTRAN     660
SUB1                       yes       FORTRAN     440
SUB2                       yes       FORTRAN     468
SHARE#DEBUG                no        Image       0
SHARE#DBGSSISHR           no        Image       0
SHARE#LIBRTL               no        Image       0
SHARE#FORRTL               no        Image       0
SHARE#LBRSHR               no        Image       0
SHARE#MTHRTL               no        Image       0
SHARE#PLIRTL               no        Image       0
SHARE#SCRSHR               no        Image       0
SHARE#SMGSHR               no        Image       0

total modules: 12.          remaining size: 53336.
```

Note that the `SHOW MODULE` command displays the remaining size of the debugger's storage area. You can increase this storage area by using the `ALLOCATE` command or the `/ALLOCATE` qualifier of the `SET MODULE` command.

In addition to using the debugger `SET MODULE` command, you may also need to use the debugger `SET SCOPE` command to reference symbols during the debugging session; see Section 17.6.3.2.

17.6.2 Referencing Locations in a Program

As illustrated in previous sections, the debugger allows you to specify addresses as symbolic names in debugger commands. For example, to examine a variable you simply refer to it by its name; you do not need to concern yourself with its actual memory location. This form of symbolic expression applies to data addresses, such as variables and array elements, and to program addresses, such as program labels and program unit names.

When you are debugging more than one program unit, you should be aware of the concept of scope because it affects how the debugger searches for symbols (see Section 17.6.3). The following sections describe how to specify data and program addresses in debugger commands.

17.6.2.1 Specifying Data Addresses

You can specify data addresses symbolically using the same syntax as in your FORTRAN program. For example:

```
DBG> DEPOSIT ISSUE=100
```

```
DBG> EXAMINE PURCH(I,J+1)
```

The first command deposits a value of 100 in the program variable `ISSUE`, and the second command examines an element of array `PURCH`.

You can reference array elements with constants and variable expressions. If you reference a variable or array element that is not defined in the symbol table, or if you attempt to reference out of the array bounds defined at compile time, the debugger issues a warning.

17.6.2.2 Current, Previous, and Next Locations

The debugger provides a quick method for referencing any of three relative data addresses, or locations:

- The current location (the location most recently referenced by an `EXAMINE` or `DEPOSIT` command)
- The previous location (the location at the next lower address from the current location)
- The next location (the location at the next higher address from the current location)

To specify the current location, type a period (`.`). For example:

```
DBG> DEPOSIT .=100
```

This command puts a value of 100 in the current location.

To specify the previous location, type an up-arrow or a circumflex (`^`). For example:

```
DBG> EXAMINE ^
```

This command displays the the contents of the previous location.

To specify the next higher location, simply omit the address value entirely. For example:

```
DBG> EXAMINE
```

This command displays the next location's contents.

17.6.2.3 Specifying Program Addresses

You can specify program addresses by program unit name, line number, statement label, or (nonsymbolic) virtual address. To specify a program unit by name, give the command followed by the name of the program unit. For example, the command

```
DBG> SET BREAK SUB1
```

sets a breakpoint at the entry to program unit SUB1.

To specify a line number, use the %LINE prefix, as shown here:

```
DBG> SET BREAK %LINE 6
```

This command sets a breakpoint at line 6, corresponding to the compiler-generated line number shown in the listing. Note that the debugger does not recognize all line numbers. In particular, it does not recognize those line numbers associated with nonexecutable statements. If you specify such a line number, the debugger responds with a message indicating that no such line exists.

You can also set a break at a line number within a particular program unit. For example, to stop execution at line 11 in SUB2, you could set a breakpoint as follows:

```
DBG> SET BREAK SUB2\%LINE 11
```

To specify a statement label, use the %LABEL prefix. For example:

```
DBG> SET BREAK %LABEL 7
```

This command sets a breakpoint at statement label 7.

To specify a virtual address, issue the command without a prefix. For example:

```
DBG> SET BREAK 700
```

You can also enter virtual addresses in symbolic form. To do so, you must have previously defined them symbolically with the DEFINE command (see Section 17.6.4).

17.6.3 Making Symbolic References Unique—Prefixes and Scope

If the program you are debugging consists of more than one program unit, you must be sure that your symbolic references are unambiguous. Most of the time, you can let the debugger define scope for you. At certain times, however, you must give the debugger additional information in order to enable it to resolve symbolic references. For example, assume that you are debugging two program units; both units use the variable I, and both occurrences are defined in the symbol table. Unless you explicitly specify scope, the debugger may be unable to determine which variable I you want.

When you begin a debugging session, the debugger automatically defines the first program unit linked (normally the main program unit) as the default scope. However, this default scope is dynamic; that is, as you debug your program, the default scope (PC scope) is always the subroutine you are currently executing. The debugger default scope rules are as follows:

- If the symbol name is unique within the debugger symbol table, the debugger can reference its definition.
- If the symbol is not unique within the symbol table, but is used within the current scope, the debugger uses the definition for the symbol as defined by the current scope.
- If the symbol is not defined within the symbol table, the debugger issues a message indicating that the symbol is “not in the symbol table.” In this case, you might have misspelled the symbol, forgotten to use the SET MODULE command to include the symbols of a particular module, or forgotten the /DEBUG qualifier when you compiled or linked the program.

If the symbol is not unique within the program and is not used within the current scope, the debugger issues a message indicating that the symbol “is not unique.” In this case, you must specify a pathname or use the SET SCOPE command, as shown in the following sections, to resolve the ambiguity for the debugger. (If necessary, use the SHOW SYMBOL command to list the modules that define the symbol.)

17.6.3.1 Pathname Prefix

You can make a symbol unique by specifying a string of symbolic names connected by backslashes that fully identify the symbol. The string, or pathname, can include the symbolic name of the subroutine, block, labeled section, and/or the number of the line that contains it. The pathname can be incomplete, so long as it makes the symbol unique. Usually one pathname prefix is sufficient to make the symbol unique. For example, while executing within subroutine SUB2, the following command examines the variable M in subroutine SUB1:

```
DBG> EXAMINE SUB1\M
```

Examples of other pathnames are:

```
DBG> EXAMINE %LINE 2\M  
DBG> EXAMINE SUB1%LINE 3\M  
DBG> EXAMINE 0\M
```

Subroutines in pathnames can be specified numerically (as shown in the preceding example), where the currently executing subroutine is 0, the subroutine that calls the currently executing subroutine is 1, the subroutine that calls the subroutine that calls the currently executing subroutine is 2, and so on.

17.6.3.2 SET SCOPE Command

You can use the SET SCOPE command to specify one or more program regions to be used by default in the interpretation of symbols. For example, to make the subroutine SUB1 the default scope, specify:

```
DBG> SET SCOPE SUB1
```

Subsequent references to symbols without pathnames use the pathname string "SUB1\" as the default pathname prefix.

```
DBG> EXAMINE M
SUB1\M:          0
```

You can also use a list of subroutines or pathnames as a parameter of the SET SCOPE command to set the order in which the debugger searches for the symbol referenced. For example, the following command makes the debugger search first the subroutine SUB1 and then search the subroutine SUB2 for whatever symbol is being referenced.

```
DBG> SET SCOPE SUB1,SUB2
DBG> SHOW SCOPE
scope: SUB1, SUB2
```

17.6.4 Defining Addresses Symbolically

You can assign a symbolic name to a program location, value, or character string with the debugger command DEFINE. You might, for instance, define a symbol to represent a frequently referenced location that is hard to remember. The following example assigns the symbolic name M2 to the integer variable M in the subroutine SUB2 and then references the variable by its assigned name:

```
DBG> DEFINE M2 = SUB2\M
DBG> EXAMINE M2
SUB2\M: 6
```

The symbol definition lasts for the duration of the debugging session or until you cancel it with the UNDEFINE command.

17.6.5 Displaying Symbol Information—SHOW SYMBOL

To display information about the symbols in your program, use the SHOW SYMBOL command. For example, to display the address and type of all symbols named M, enter:

```
DBG> SHOW SYMBOL/ADDRESS/TYPE M
data SUB2\M
  address: 00000250
  atomic type, longword integer, size: 4 bytes
data SUB1\M
  address: .(.%AP+4)
  atomic type, longword integer, size: 4 bytes
```

To display information about symbols you have defined during the debugging session, use the `SHOW SYMBOL/DEFINED` command:

```
DBG> SHOW SYMBOL/DEFINED M2
defined M2
    bound to: SUB2\M
    was defined /address
```

By default, the `SHOW SYMBOL` command returns information about global symbols and symbols in those subroutines that have been set (either by default or with the `SET MODULE` command). The `IN` clause allows you to restrict the `SHOW SYMBOL` command to one or more subroutines. You can use the wildcard character (an asterisk) with the `SHOW SYMBOL` command to match any number of characters in the symbol's name. The following command displays information about all symbols within the scope of the subroutine `SUB2` (any specified scope must be in a module that is set in order for the `SHOW SYMBOL` command to work properly):

```
DBG> SHOW SYMBOL * IN SUB2
routine SUB2
data SUB2\N
data SUB2\M
data SUB2\K
data SUB2\L
```

To display the address specification of all symbols beginning with the characters `XYZ`, specify:

```
DBG> SHOW SYMBOL/ADDRESS XYZ*
```

17.7 Examining and Manipulating Data

You use the `EXAMINE` and `EVALUATE` commands and the `DEPOSIT` command to, respectively, examine and manipulate data as your program executes. If the locations referenced by the following commands are not in your default scope, you must set the module, as described in Section 17.6.1, and specify a scope or pathname, as described in Section 17.6.3. (For all examples in this section, it is assumed that the module and scope are set properly.)

Table 17-2 summarizes the command qualifiers of particular significance in FORTRAN debugging for the `EXAMINE`, `EVALUATE`, and `DEPOSIT` commands.

Table 17-2: Debugger Command Qualifiers

Qualifier	Function	Commands
/ADDRESS	Indicates that an address value is desired	EVALUATE
/HEXADECIMAL /OCTAL	Override the default radix (decimal)	EVALUATE EXAMINE DEPOSIT
/BYTE /WORD /ASCII /LONG /FLOAT /D_FLOAT /G_FLOAT /H_FLOAT	Override the EXAMINE display type	EXAMINE

17.7.1 Hints about the Use of Expressions

When using the EXAMINE, EVALUATE, and DEPOSIT commands, be aware of the difference between an address expression and a language expression:

- **Address expression**—an address expression specifies a program location. If the location is that of a symbol defined by the source program, it has a language-dependent data type associated with it; otherwise, no data type is associated with it. An address expression may consist of a single operand or multiple operands combined with the debugger operators (see Table 17-3); it is evaluated as follows:
 1. Parenthesized parts of the expression
 2. Operators by rank low to high (see Table 17-3)
 3. Operators of the same rank from left to right

The result of an address expression is a 32-bit longword integer that represents a program location. In Table 17-1, arguments named “address-expression” should be specified as address expressions.

- **Language expression**—a language expression specifies a value; the value is associated with a data type. A language expression can consist of a single operand or multiple operands combined with operators; the expression is evaluated according to

the rules of precedence for the source language. (Section D.1 lists the legal FORTRAN operators and FORTRAN's rules of precedence.) The result of a language expression must be a value that is valid for the current source language. In Table 17-1, arguments named "expression" accept language expressions.

For example, assume that you have a symbol named NUMBER and that it has a value of 3 and is located at address 1600. The EXAMINE command, which expects an address expression, interprets (NUMBER + 1) as 1601. The EVALUATE command, which expects a language expression, interprets (NUMBER + 1) as 4.

Table 17-3: Debugger Operators

Operator	Rank	Description
. or @	1	Unary operators specify the contents of the operand.
+ or -	1	Unary operators specifying the positive or negative value of the operand.
* or /	2	Binary operators specifying the multiplication or division of the operands.
+ or -	3	Binary operators specifying the addition or subtraction of the operands.

Array element references and record field references are also supported in address and language expressions.

17.7.2 Displaying Values—EXAMINE

The EXAMINE command displays the contents of a specified program location. For example, to display the contents of the variable M in the routine SUB2, specify:

```
DBG> EXAMINE M
SUB2\M: 6
```

To display array elements, specify the elements individually (1:1, 2:2, and so forth), in a range (1:10), or with a wildcard (*). The following command displays three elements of the array ARRMN:

```
DBG> EXAMINE ARRMN(12:14)
MAIN\ARRMN
  (12):      14.50000
  (13):      14.50000
  (14):      14.50000
```

17.7.3 Calculating Values—EVALUATE

The EVALUATE command displays the value of a specified language expression. For example, to add the value in ARRMN(12) and the value in ARRMN(17), where both symbols are defined in the program unit MAIN, specify:

```
DBG> EVALUATE ARRMN(12) + ARRMN(17)
29.00000
```

You can also perform arithmetic calculations with the EVALUATE command that may or may not be related to your program, in effect using the debugger as a calculator.

17.7.4 Assigning Values—DEPOSIT

The DEPOSIT command assigns a language expression to a program location. For example, to assign the value 9 to the array element ARRMN(19) in program unit MAIN, specify:

```
DBG> DEPOSIT ARRMN(19) = 9.
DBG> EXAMINE ARRMN(19)
MAIN\ARRMN(19): 9.000000
```

17.7.5 Specifying Data Type

Typically, when you examine a program location, you want to use the data type that your program has associated with that location. For example, if you have defined STATUS as a variable of data type INTEGER, when you examine STATUS in the debugger, you probably want to examine it as an integer value. By default, the debugger uses the program assigned data types when it displays program locations.

However, if necessary, you can specify a data type for a program location other than the data type your program has associated with it:

- SET TYPE/OVERRIDE command—sets the default data type for debugger commands that interpret and display program data. For example, if you set the default data type to be BYTE, any variable that you examine (regardless of how you declared it in your program) will be displayed as a BYTE value. In the following example, the program unit MAIN declared PI as a REAL*4 value.

```
DBG> EXAMINE PI
CIRCLE\PI:      3.141593
DBG> SET TYPE/OVERRIDE BYTE
DBG> EXAMINE PI
CIRCLE\PI:      73
```

- Data type qualifiers on EXAMINE command—indicates that the modified command should display or evaluate the referenced location using the data type specified by the qualifier. A type qualifier overrides the default type specified with the SET TYPE/OVERRIDE command. In the following example, the default type for the debugging session is set to BYTE. The EXAMINE command uses the /FLOAT qualifier (FORTRAN data type REAL*4) to examine the variable PI.

```
DBG> SET TYPE/OVERRIDE BYTE
DBG> EXAMINE PI
CIRCLE\PI:      73
DBG> EXAMINE/FLOAT PI
CIRCLE\PI:      3.141593
```

The following table displays debugger data types and their FORTRAN equivalents. The SHOW TYPE command displays the default FORTRAN data type and the SHOW TYPE/OVERRIDE displays the current type setting of the /OVERRIDE qualifier.

Debugger	FORTRAN
BYTE	LOGICAL*1
WORD	INTEGER*2, LOGICAL*2
LONG	INTEGER*4, LOGICAL*4
FLOAT	REAL*8
D_FLOAT	REAL*8
G_FLOAT	REAL*8
H_FLOAT	REAL*8
ASCII[:n]	CHARACTER*n

In addition to the familiar data types, the debugger provides an INSTRUCTION data type, which interprets values as VAX machine code instructions. The INSTRUCTION data type is a powerful debugging tool for those programmers who are familiar with machine code instructions.

17.7.6 Specifying Radix

To specify a radix other than the decimal default, use either the SET MODE command or include a radix qualifier (for example, OCTAL or HEX) with individual debugger commands.

- **SET MODE** command—sets the default radix and symbolic mode for debugger commands that display and interpret data. The default for FORTRAN is decimal radix and symbolic mode (displaying symbolic rather than numeric addresses). For example, the following commands display the contents of R0 in hexadecimal and then in octal as the default radix mode:

```
DBG> SET MODE HEX
DBG> EXAMINE R0
CIRCLE\R0:      00004410
DBG> SET MODE OCTAL
DBG> EXAMINE R0
CIRCLE\R0:      00000042020
```

- **Radix qualifier on EXAMINE** command—controls the radix of values interpreted by individual commands (and whether those commands display symbolic or numeric addresses). A qualifier that specifies radix overrides the default radix set by the SET MODE command, as is shown in the following example:

```
DBG> EXAMINE AREA
CIRCLE\AREA:    201.0619
DBG> EXAMINE/HEX AREA
CIRCLE\AREA:    0FDB4449
```

To display the current type and mode settings, use the **SHOW TYPE** and **SHOW MODE** commands. For example, the following commands display the FORTRAN default type and mode:

```
DBG> SHOW TYPE
type: long integer
DBG> SHOW MODE
mode: symbolic, noscreen, keypad
input radix: decimal
output radix: decimal
```

17.7.7 Using Numeric Data Types in Expressions

You can use arrays and records and any of the following types of values in expressions in all appropriate debugger commands:

```
LOGICAL*1
LOGICAL*2
LOGICAL*4
INTEGER*2
INTEGER*4
REAL*4
REAL*8
REAL*16
COMPLEX*8
COMPLEX*16
CHARACTER
```


You can use the EXAMINE and DEPOSIT commands with any of the VAX FORTRAN data types. Furthermore, if you attempt to deposit a numeric value into a variable or array element that does not have a matching data type, the value is converted to the data type of the variable or array element.

To deposit a complex value, you must use two DEPOSIT commands:

```
DEPOSIT x = real part
DEPOSIT/FLOAT x = imaginary part
```

For example:

```
DBG> DEPOSIT CPLX=3.4
DBG> DEPOSIT/FLOAT CPLX=-4.7
```

When you examine a complex variable or array element, the data is displayed as a complex constant, as (real part, imaginary part).

When you deposit a real number, you must specify a decimal point. To distinguish REAL*4, REAL*8, and REAL*16 numbers, use E, D, and Q, respectively. For example:

Number	Data Type
24.1	REAL*4 (default)
24.1E0	REAL*4
24.1D0	REAL*8
24.1Q0	REAL*16
241E0	Invalid (no decimal point)

17.8 Using Screen Displays

Screen mode debugging allows you to keep various types of debugging information on the screen by dividing the screen into sections and displaying a different type of information in each section. The sections of the screen are called windows and the contents of the windows are called displays. In screen mode, the debugger defines a number of default windows and maintains three default displays:

- A display of source lines (SRC)
- A display of debugger output (OUT)
- A display of register contents (REG)

The bottom lines on the screen are reserved for debugger commands. (These lines also receive program output, unless the program specifies otherwise.)

17.8.1 Invoking and Terminating Screen Mode

To use screen mode debugging, press the PF3 key or enter the SET MODE SCREEN command. Two displays appear on the screen by default:

- SRC, the default source display, appears in window H1 (the top half of your screen). The SRC display, by default, points to the next executable source line and shows the four lines preceding and following it. The entire source program is available through scrolling, as long as the conditions for normal source display are met (see Section 17.8.3.2 for a list of these conditions).
- OUT, the default output display, appears in window H2 (the bottom half of your screen). The OUT display, by default, shows debugger output, such as responses to SHOW and EXAMINE commands. (The 100 most recent lines of debugger output are available through scrolling.)

The display name and characteristics are placed on the title line of the window.

The following screen appears when you execute a STEP command followed by a SET MODE SCREEN command when debugging program unit MAIN:

```
-SRC: module MAIN-source-scroll-----
  1:          PROGRAM MAIN
  2:          INTEGER I/1/, J/11/, K/21/
  3:          REAL ARRMN(10:20)/11*14.5/
->  4:          CALL SUB1(I)
  5:          CALL SUB2(J)
  6:          END

-OUT-output-----
stepped to MAIN
  4:          CALL SUB1(I)

-----
DBG>SET MODE SCREEN
DBG>STEP
DBG>
```

Note: The arrow at the left side of the SRC display indicates the next statement to be executed.

The debugger makes the third display, REG, available but not visible by default; the REG display shows the current contents of machine registers.

To see existing screen displays (including those not currently displayed on the screen), specify the `SHOW DISPLAY` command. The information is displayed in the `OUT` display area.

```
-SRC: module MAIN-source-scroll-----
-> 1:      PROGRAM MAIN
   2:      INTEGER I/1/, J/11/, K/21/
   3:      REAL ARRMN(10:20)/11*14.5/
   4:      CALL SUB1(I)
   5:      CALL SUB2(J)
   6:      END
```

```
-OUT-output-----
display SRC at H1, size = 50
  kind = SOURCE (EXAMINE/SOURCE .0)
display REG at R2, size = 5, removed, kind = REGISTER
display OUT at H2, size = 100, kind = NORMAL
```

```
-----
DBG>SET MODE SCREEN
DBG>SHOW DISPLAY
DBG>
```

When debugger output from a single command exceeds the dimensions of the `OUT` display (the output from the `SHOW WINDOW` command, for example), the beginning of the display is not visible. To view the entire display, you can scroll the display (see Section 17.8.3.1), place the display into window `FS` (full screen) or another large screen region (see Section 17.8.3.2) or terminate screen mode (`SET MODE NOSCREEN`) before entering the command. (A subsequent `SET MODE SCREEN` command restores the screen displays.)

17.8.2 Defining Windows

The debugger provides a number of default windows that allow you to treat the entire screen as a single window (`FS`) or divide the screen into halves (`H1,H2`), into thirds (`T1,T2,T3`), or into quarters (`Q1,Q2,Q3,Q4`). In addition, the debugger defines a number of windows that combine the fractional screens; for example, window `Q12` refers to the top two quarters of the screen (same as `H1`) and window `T23` refers to the bottom two thirds of the screen. Most of the debugger windows hold any type of display; however, three windows (`R1,R2,R3`), each one third of the screen, are reserved for register displays.

A window is defined by its top line and the number of lines that it can hold. For example, the window `H1` is defined as `H1 (1,9)` and `H2` as `H2 (11,9)`. You can use the `SET WINDOW` command to define your own windows; however, with all of the default windows, your own definitions are usually superfluous.

Use the `SHOW WINDOW` command to display the name and dimensions of all windows currently defined. Use the `CANCEL WINDOW` command to delete one or more windows.

17.8.3 Manipulating Displays

You can manipulate screen displays in several ways, including showing them on the screen, scrolling forward or backward through them, and removing them from the screen.

You can use the following pseudo-display names to reference displays in debugger commands:

<code>%CURDISP</code>	The current (most recently viewed) display
<code>%NEXTDISP</code>	The next display in the list
<code>%NEXTOUTPUT</code>	The next output display
<code>%NEXTSCROLL</code>	The next scrolling display
<code>%NEXTSOURCE</code>	The next source display

17.8.3.1 Scrolling Screen Displays

The `SCROLL` command allows you to show different parts of a display on the screen. The `SELECT/SCROLL` command determines which display is affected by the `SCROLL` command. For example, to establish the `SRC` display as the default scrolling display, specify:

```
DBG> SELECT/SCROLL SRC
```

The `SRC` display remains the default parameter of the `SCROLL` command until you specify another `SELECT/SCROLL` command. Using key 3 on the keypad, you can step through the scrolling displays until you get the display that you want.

To display the previous 5 lines of source code, you could specify:

```
DBG> SCROLL/UP:5
```

Typically, however, you scroll a display using the keypad keys:

- Up—key 8 scrolls towards the beginning of the display by entering the `SCROLL/UP` command. To scroll to the top of the display, press PF1 followed by key 8.
- Down—key 2 scrolls towards the end of the display by entering the `SCROLL/DOWN` command. To scroll to the bottom of the display, press PF1 followed by key 2.
- Left—key 4 scrolls towards the left of the display by entering the `SCROLL/LEFT` command.
- Right—key 6 scrolls toward the right of the display by entering the `SCROLL/RIGHT` command.

Keypad key 5 refreshes the current source display (`SRC`, by default), causing the next source line to be executed (marked with an arrow at the left of the source display) to appear in the middle of the display.

17.8.3.2 Creating Screen Displays

In addition to the default displays, you can define other source, output, and register displays. To create a display, use the `SET DISPLAY` command:

```
SET DISPLAY display-name [AT window] [type]
```

You must specify a display name; optionally, you can specify the window into which the display is mapped and the type of display to create. The following display types are available:

- **DO (command-list)**—display contains the results of the debugger commands specified in the command list. The command list is executed each time the debugger regains control. If you specify more than one command, separate the commands using semicolons.
- **NORMAL**—display contains all debugger output, but only if the display is selected for output with the **SELECT/OUTPUT** command. By default, the **OUT** display is selected for output.
- **REGISTER**—display contains the VAX registers and their contents; the display is updated each time the debugger regains control. The **REG** display is type **REGISTER**.
- **SOURCE**—display contains the program source statements, but only if the display is selected for source display with the **SELECT/SOURCE** command.
- **SOURCE (command-list)**—display contains the results of the debugger commands specified in the command list, but only if the display is selected for source display with the **SELECT/SOURCE** command. The command list, which should consist of a single **TYPE** or **EXAMINE/SOURCE** command, is executed each time the debugger regains control. The **SRC** display is type **SOURCE** with a command list of **EXAMINE/SOURCE .0\%PC** (examine the source line in the current module at the location that is in the program counter). By default, the **SRC** display is selected for source display.

The following example defines a display **XYZ** to be shown in window **T2** on the screen. Each time the debugger regains control, it executes the **DO** command list (here the **EXAMINE** command) and lists the results in display **XYZ**:

```
-SRC: module CIRCLE-source-scroll-----
-> 1:      PROGRAM CIRCLE
   2:      DO I = 1,3,1
   3:  4    TYPE 5
   4:  5    FORMAT ( ' enter radius value ' )
   5:      ACCEPT 10,RADIUS
   6: 10    FORMAT (F6,2)
-XYZ-----
CIRCLE\AREA:      0.0000000
CIRCLE\RADIUS:    0.0000000

-OUT-output-----

-----
DBG>SET DISPLAY XYZ AT T2 DO (EXAMINE AREA,RADIUS)
DBG>SET BREAK %LINE 9
DBG>
```

17.8.3.3 Accessing Displays

To control the display of different screen displays, you can use keypad keys or the DISPLAY command. Typically, the keypad keys (key 9 and key minus) are used for this operation:

- Key 9 performs the DISPLAY %NEXTDISP command; it displays the next display on the debugger's current list. Repeated use of this key causes the debugger to step through the displays in the display list.
- Key minus (-) performs the DISPLAY %NEXTDISP AT FS and SELECT/SCROLL %CURDISP commands. It displays the next display in the display list on the full screen, covering any other displays that are currently visible. It also establishes this new display as the current scrolling display. Repeated use of this key allows you to step through your displays, displaying each one at window FS (the full screen).

17.8.3.4 Removing Screen Displays

You can use the CANCEL DISPLAY command to delete displays that you previously set.

17.9 Sample Debugging Sessions

This section contains two FORTRAN programs and the listings (with comments) of sample debugging sessions that control the execution and analyze the effects of the programs. The second debugging session (Section 17.9.2) is particularly important because it shows how to access symbols in a multi-unit program.

17.9.1 Debugging a FORTRAN Program Unit

Figure 17-2 illustrates a program that requires debugging. The program was compiled and linked without diagnostic messages from either the compiler or the linker. (Appendix E summarizes compiler diagnostic messages.) However, the program produces erroneous results because of a missing asterisk in the exponentiation operator (RADIUS*2 should be RADIUS**2). This error is so obvious that you hardly need the services of the debugger to find it. However, for purposes of illustration, this example deals with the error as though it were not obvious.

```
0001          PROGRAM CIRCLE
0002          DO I = 1,3,1
0003             TYPE 5
0004          5   FORMAT ( ' enter radius value ' )
0005             ACCEPT 10,RADIUS
0006          10  FORMAT (F6.2)
0007             PI = 3.1415927
0008             AREA = PI*RADIUS*2
0009             TYPE 15, AREA
0010          15  FORMAT ( ' area of circle equals ',F10.3)
0011             END DO
0012             STOP
0013             END
```

Figure 17-2: Sample FORTRAN Program

The key to debugging is to find out what happens at critical points in your program. To do this, you need a way to stop execution at these points and look at the contents of program variables to see if they contain the correct values. Points at which execution is stopped are called breakpoints. The SET BREAK command lets you specify where you want to stop the program. You can specify a breakpoint at the beginning of a routine or at a specific line.

To look at the contents of a location, use the EXAMINE command. To resume execution, use either the GO or STEP command. The DEBUG commands relevant to FORTRAN are discussed in subsequent sections of this chapter.

Figure 17-3 is an example of typical terminal dialog for a debugging session. The circled numbers are keyed to notes that follow the figure and explain the dialog.

```

$ FORTRAN/LIST/NOOPTIMIZE/DEBUG CIRCLE ❶
$ LINK/DEBUG CIRCLE ❷
$ RUN CIRCLE

      VAX DEBUG Version 4.1-2

%DEBUG-I-INITIAL, language is FORTRAN, module set to 'CIRCLE' ❸
DBG> SET BREAK %LINE 8 ❹
DBG> GO ❺
enter radius value
24.
break at CIRCLE%\%LINE 8 ❻
      B:      AREA = PI*RADIUS*2
DBG> EXAMINE PI ❼
CIRCLE\PI:      3.141593
DBG> EXAMINE RADIUS ❸
CIRCLE\RADIUS:  24.000000
DBG> EXAMINE AREA ❹
CIRCLE\AREA:    0.0000000
DBG> GO ❽
area of circle equals      150.796
enter radius value
3.
break at CIRCLE%\%LINE 8
      B:      AREA = PI*RADIUS*2
DBG> CANCEL BREAK %LINE 8 ❶
DBG> GO
area of circle equals      18.850
enter radius value
13.
area of circle equals      81.681
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion' ❿
DBG> EXIT ⓫
$

```

Figure 17-3: Sample Debugging Session

- ❶ Invoke the FORTRAN compiler, specifying the qualifiers shown. You should include the /NOOPTIMIZE qualifier when you use symbolic debugging.
- ❷ Link your program using the /DEBUG qualifier to include a symbol table for the debugger.
- ❸ In response to the RUN command, the debugger displays its identification, indicating that your program will be executed under the debugger's control. Following the identification message, the debugger displays an initial message, identifying the language and module settings it has assumed. The debugger derives these mode settings from the first module specified in the LINK command. If this message does not appear, or if the settings assumed are not appropriate, use the SET LANGUAGE and SET MODULE commands.
- ❹ Set a breakpoint at an appropriate point in the program. This point should be one at which you are able to examine key variables. Note: Breakpoints suspend execution just before the point specified.
- ❺ Begin program execution. The debugger displays the point at which execution starts.
- ❻ The debugger announces that it has suspended execution at the specified breakpoint.
- ❼ Examine the variable PI to make sure that the correct value is stored there. The debugger displays the contents of PI, showing that its scope is in module CIRCLE.
- ❽ Examine the variable RADIUS. The debugger shows that the specified value has been properly stored.
- ❾ Examine the variable AREA.
- ❿ Resume execution. The debugger displays a message indicating the point of program resumption.
- ⓫ Cancel the breakpoint at line 8 so that subsequent calculations will not be interrupted.
- ⓬ Successful completion of the program is indicated by this message. However, as you can see, the results are incorrect.
- ⓭ Exit from the debugger.

By examining the variables PI, RADIUS, and AREA as the program is executing, you can determine that the correct values are being stored. It follows, then, that the error is probably in the expression of the formula for computing the area. To correct the problem, you must edit and recompile the source program, with the exponentiation operator properly specified in the formula expression.

17.9.2 Debugging a FORTRAN Program with Subprograms

This section shows how you can use some of the debugging commands and concepts discussed in the preceding sections, and gives the responses you might expect from the debugger. Figure 17-4 illustrates a program with three program units:

```
0001      PROGRAM MAIN
0002      INTEGER I/1/, J/11/, K/21/
0003      REAL ARRMN(10:20)/11*14.5/
0004      CALL SUB1(I)
0005      CALL SUB2(J)
0006      END
0001      SUBROUTINE SUB1(M)
0002      INTEGER M,N/5/, L/31/
0003      M = M + 1
0004      RETURN
0005      END
0001      SUBROUTINE SUB2(N)
0002      INTEGER M/6/, N,K/41/, L/51/
0003      N = N + 1
0004      RETURN
0005      END
```

Figure 17-4: Sample FORTRAN Program with Subprograms

Figure 17-5 is the sample debugging session. It illustrates in particular how to specify addresses to the debugger when several program units are involved in the session and when a symbol is not unique in the symbol table. The circled numbers are keyed to explanatory notes following the session.

VAX DEBUG Version 4.1-2

```
%DEBUG-I-INITIAL, language is FORTRAN, module set to 'MAIN' ❶
DBG> EXAMINE ARRMN(1) ❷
%DEBUG-I-SUBOUTBND, subscript 1 is out of bounds; value is 1, bounds are
10:20
%DEBUG-W-NOACCESSR, no read access to virtual address 000001DC
DBG> EXAMINE ARRMN(18):ARRMN(K-I) ❸
MAIN\ARRMN(18): 14.50000
MAIN\ARRMN(19): 14.50000
MAIN\ARRMN(20): 14.50000
DBG> EXAMINE L ❹
%DEBUG-W-NOSYMBOL, symbol 'L' is not in the symbol table
DBG> SHOW MODULE ❺
```

module name	symbols	language	size
MAIN	yes	FORTRAN	660
SUB1	no	FORTRAN	440
SUB2	no	FORTRAN	468
SHARE\$DEBUG	no	Image	0
SHARE\$DBGSSISHR	no	Image	0
SHARE\$LIBRTL	no	Image	0
SHARE\$FORRTL	no	Image	0
SHARE\$LBRSHR	no	Image	0
SHARE\$MTHRTL	no	Image	0
SHARE\$PLIRTL	no	Image	0
SHARE\$SCRSHR	no	Image	0
SHARE\$SMGSHR	no	Image	0

total modules: 12. remaining size: 53664.

```

DBG> SET MODULE/ALL
DBG> EXAMINE L
%DEBUG-W-NOUNIQUE, symbol 'L' is not unique
DBG> EXAMINE SUB1\L
SUB1\L: 31
DBG> SET SCOPE SUB1
DBG> EXAMINE N
SUB1\N: 5
DBG> EXAMINE M
%DEBUG-W-SYMNOTACT, symbol SUB1\M not active or not in active scope
DBG> SET BREAK %LINE 3
DBG> SHOW BREAK
breakpoint at SUB1\%LINE 3
DBG> GO
break at SUB1\%LINE 3
3:      M = M + 1
DBG> EXAMINE I
MAIN\I: 1
DBG> STEP
stepped to SUB1\%LINE 4
4:      RETURN
DBG> EXAMINE .
MAIN\I: 2
DBG> EXIT
$

```

Figure 17-5: Sample Multiunit Debugging Session

- ❶ The debugger indicates that the language is FORTRAN and the module is set to program unit MAIN (that is, symbols contained in MAIN have been placed in the symbol table).
- ❷ Attempt to examine an out-of-bounds array element. The debugger issues a warning.
- ❸ Examine contents of array elements 18 through 20 (expression K-I). The debugger prints their values.

- ④ Attempt to examine the variable L. However, the debugger announces that this symbol is not in the symbol table.
- ⑤ Use SHOW MODULE to see which program units have symbols in the symbol table.
- ⑥ Place all symbols in the symbol table.
- ⑦ Examine variable L again. The debugger indicates that this symbol is duplicated in the symbol table.
- ⑧ Explicitly specify variable L in program unit SUB1. The debugger successfully displays the contents of L.
- ⑨ Examine the variable N after issuing the SET SCOPE command. Program unit SUB1 is now the scope and N is examined.
- ⑩ Attempt to examine a dummy argument. This produces an informational message from the debugger, warning that the symbol M is not within the PC scope. Because it is only correct to examine a dummy argument within the routine declaring it, the value displayed for M is unpredictable.
- ⑪ Set a breakpoint at program address %LINE 3 and begin program execution (GO). The debugger identifies the starting point and breakpoint. Execution stops at line 3 in SUB1.
- ⑫ Examine variable I located in program unit MAIN. (This symbol is unique in the symbol table and so is examined successfully even though the scope is set to SUB1.)
- ⑬ Step to line 4 (RETURN) in SUB1 check the value of the current location. Notice that the previous EXAMINE command established the variable I in routine MAIN as the current location.

Chapter 18

Error Processing

During execution, your program may encounter errors or exception conditions. These conditions can result from errors that occur during I/O operations, from invalid input data, from argument errors in calls to the mathematical library, from arithmetic errors, or from system-detected errors. The Run-Time Library provides default processing for error conditions, generates appropriate messages, and takes action to recover from errors whenever possible. You can, however, explicitly supplement or override default actions by using:

- The error (ERR) and end-of-file (END) specifiers in I/O statements to transfer control to error-handling code within the program.
- The I/O status specifier (IOSTAT) in I/O statements to identify FORTRAN-specific errors based on the value of IOSTAT.
- The VAX condition-handling facility (including user-written condition handlers) to tailor error processing to the special requirements of your applications. Note: Information about user-written condition handlers is provided in the *VAX FORTRAN User's Guide*.

These error-processing methods are complementary; you can use them within the same program. However, before attempting to write a condition handler, you should be familiar with the VAX condition-handling facility and with the condition-handling description in the *VAX/VMS Run-Time Library Reference Manual* and the *VAX FORTRAN User's Guide*.

This chapter describes how the Run-Time Library processes errors. It also provides information about using I/O specifiers for explicit error processing and control, and shows how these methods affect the default error processing of the Run-Time Library.

18.1 Run-Time Library Default Error Processing

The Run-Time Library contains condition handlers that process a number of errors that may occur during FORTRAN program execution. A default action is defined for each FORTRAN-specific error recognized by the Run-Time Library. The default actions described throughout this chapter occur unless overridden by explicit error-processing methods.

How the Run-Time Library actually processes errors depends upon several factors: how severe the error is, whether an I/O error-handling specifier or a condition handler was used, and whether the error permits continuation.

Table 18-1 lists the FORTRAN-specific errors processed by the Run-Time Library. For each error, the table shows the message text, the symbolic condition name, the FORTRAN-specific error code, and the severity code. (Note: Refer to Table E-4 for more detailed descriptions of errors processed by the Run-Time Library.)

The condition symbols shown in the left column are the status codes signaled by the FORTRAN RTL I/O support routines. You can define these symbolic values in your program by including the module \$FORDEF from the system-supplied default library FORSYSDEF.TLB.

The error numbers shown in the second column are the standard Digital FORTRAN error numbers that are compatible with other versions of Digital FORTRAN. These are the error values returned to IOSTAT variables when an I/O error is detected. They are also used to index the error table maintained by the ERRSET and ERRTST subroutines. See Appendix B in the *VAX FORTRAN User's Guide* for descriptions of the ERRSET and ERRTST subroutines.

The codes in the third column indicate the severity of the error conditions. All FORTRAN-specific errors have severity codes of either error (E) or severe error (F). As shown in the table, most FORTRAN-specific errors are severe. If no explicit recovery action is specified for a severe error, program execution terminates by default.

The letter C in the "Severity" column of the table means that you can continue execution immediately after the error, if a user-written condition handler specifies that execution should continue. If there is no letter C in the "Severity" column, you cannot continue execution immediately after the error. If you attempt to do so, program execution terminates.

When errors occur for which no recovery method is specified, the program exits; that is, an error message is printed and execution of the program terminates. To prevent program termination, you must include an appropriate I/O error-handling specifier (see Sections 18.2 and 18.3) or a condition handler that performs an unwind (see Chapter 6 in the *VAX FORTRAN User's Guide*).

Table 18-1: Summary of FORTRAN Run-Time Errors

FORTRAN Condition Symbol	Error Number	Severity	Message Text
FOR\$_NOTFORSPE	1	F	not a FORTRAN-specific error
FOR\$_SYNERRNAM	17	F	syntax error in NAMELIST input
FOR\$_TOOMANVAL	18	F	too many values for NAMELIST variable
FOR\$_INVREFVAR	19	F	invalid reference to variable in NAME- LIST input
FOR\$_REWERR	20	F	REWIND error
FOR\$_DUPFILSPE	21	F	duplicate file specifications
FOR\$_INPRECTOO	22	F	input record too long
FOR\$_BACERR	23	F	BACKSPACE error
FOR\$_ENDDURREA	24	F	end-of-file during read
FOR\$_RECNUMOUT	25	F	record number outside range
FOR\$_OPEDEFREQ	26	F	OPEN or DEFINE FILE required
FOR\$_TOOMANREC	27	F	too many records in I/O statement
FOR\$_CLOERR	28	F	CLOSE error
FOR\$_FILNOTFOU	29	F	file not found
FOR\$_OPEFAI	30	F	open failure
FOR\$_MIXFILACC	31	F	mixed file access modes
FOR\$_INVLOGUNI	32	F	invalid logical unit number
FOR\$_ENDFILERR	33	F	ENDFILE error
FOR\$_UNIALROPE	34	F	unit already open
FOR\$_SEGRECFOR	35	F	segmented record format error
FOR\$_ATTACCNON	36	F	attempt to access non-existent record
FOR\$_INCRECLEN	37	F	inconsistent record length
FOR\$_ERRDURWRI	38	F	error during write
FOR\$_ERRDURREA	39	F	error during read
FOR\$_RECIO_OPE	40	F	recursive I/O operation
FOR\$_INSVIRMEM	41	F	insufficient virtual memory
FOR\$_NO_SUCDEV	42	F	no such device
FOR\$_FILNAMSPPE	43	F	file name specification error

Table 18-1 (Cont.): Summary of FORTRAN Run-Time Errors

FORTRAN Condition Symbol	Error Number	Severity	Message Text
FOR\$_INCRECTYP	44	F	inconsistent record type
FOR\$_KEYVALERR	45	F	keyword value error in OPEN statement
FOR\$_INCOPECLO	46	F	inconsistent OPEN/CLOSE parameters
FOR\$_WRIREAFIL	47	F	write to READONLY file
FOR\$_INVARGFOR	48	F	invalid argument to FORTRAN Run-Time Library
FOR\$_INVKEYSPE	49	F	invalid key specification
FOR\$_INCKEYCHG	50	F	inconsistent key change or duplicate key
FOR\$_INCFILORG	51	F	inconsistent file organization
FOR\$_SPERECLOC	52	F	specified record locked
FOR\$_NO_CURREC	53	F	no current record
FOR\$_REWRITERR	54	F	REWRITE error
FOR\$_DELERR	55	F	DELETE error
FOR\$_UNLERR	56	F	UNLOCK error
FOR\$_FINERR	57	F	FIND error
FOR\$_LISIO_SYN	59	F,C	list-directed I/O syntax error
FOR\$_INFFORLOO	60	F	infinite format loop
FOR\$_FORVARMIS	61	F,C	format/variable-type mismatch
FOR\$_SYNERRFOR	62	F	syntax error in format
FOR\$_OUTCONERR	63	E,C	output conversion error
FOR\$_INPCONERR	64	F,C	input conversion error
FOR\$_OUTSTAOVE	66	F	output statement overflows record
FOR\$_INPSTAREQ	67	F	input statement requires too much data
FOR\$_VFEVALERR	68	F,C	variable format expression value error
SS\$_INTOVF	70	F,C	arithmetic trap, integer overflow
SS\$_INTDIV	71	F,C	arithmetic trap, integer zero divide
SS\$_FLTTOVF	72	F,C	arithmetic trap, floating overflow
SS\$_FLTTOVF_F	72	F,C	arithmetic fault, floating overflow
SS\$_FLTDIV	73	F,C	arithmetic trap, zero divide

Table 18-1 (Cont.): Summary of FORTRAN Run-Time Errors

FORTRAN Condition Symbol	Error Number	Severity	Message Text
SS\$__FLTDIV__F	73	F,C	arithmetic fault, zero divide
SS\$__FLTUND	74	F,C	arithmetic trap, floating underflow
SS\$__FLTUND__F	74	F,C	arithmetic fault, floating overflow
SS\$__SUBRNG	77	F,C	subscript out of range
MTH\$__WRONUMARG	80	F	wrong number of arguments
MTH\$__INVARGMAT	81	F	invalid argument to math library
MTH\$__UNDEXP	82	F,C	undefined exponentiation
MTH\$__LOGZERNEG	83	F,C	logarithm of zero or negative value
MTH\$__SQUROONEG	84	F,C	square root of negative value
MTH\$__SIGLOSMAT	87	F,C	significance lost in math library
MTH\$__FLOOVEMAT	88	F,C	floating overflow in math library
MTH\$__FLOUNDMAT	89	F,C	floating underflow in math library
FOR\$__ADJARRDIM	93	F,C	adjustable array dimension error

Notes

1. The ERR transfer is taken after completion of the I/O statement for continuable errors numbered 59, 61, 63, 64, and 68; the resulting file status and record position are the same as though no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected; thus, file status and record position are undefined.
2. If no ERR address has been defined for error 63, the program continues after the error message is printed. The entire overflowed field is filled with asterisks to indicate the error in the output record.
3. Function return values for errors numbered 82, 83, 84, 87, 88, and 89 can be modified by means of user-written condition handlers. See the *VAX FORTRAN User's Guide* and the *VAX/VMS Run-Time Library Reference Manual* for information about user-written condition handlers.
4. Error number 1 (FOR\$__NOTFORSPE) indicates that an error was detected that was not a FORTRAN-specific error; that is, it was not reportable through any other message in the table. If you call ERRSNS, an error of this kind returns a value of 1. Use the fifth argument of the call to ERRSNS (condval) to obtain the unique system condition value that identifies the error. Refer to Appendix D.4.3 for more information.

Notes (Cont.)

5. If error number 93 (FOR\$_ADJARRDIM) occurs and a user-written condition handler causes execution to continue, any reference to the array in question will cause an access violation.

Refer to Table E-4 for more detailed descriptions of errors processed by the Run-Time Library.

18.2 Using the ERR and END Specifiers

When a severe error occurs during program execution, the Run-Time Library default action is to print an error message and terminate the program. You can use the ERR and END specifiers in I/O statements to override this default by transferring control to a specified point in the program. No error message is printed, and execution continues at the designated statement. For example, assume that a program contains this WRITE statement:

```
WRITE (8,50,ERR=400)
```

If an error occurs during execution of this statement, the Run-Time Library transfers control to the statement at label 400. Similarly, you can use the END specifier to handle an end-of-file condition that might otherwise be treated as an error. For example:

```
READ (12,70,END=550)
```

You can also specify ERR as a keyword in an OPEN, CLOSE, or INQUIRE statement. For example:

```
OPEN (UNIT=10, FILE='FILNAM', STATUS='OLD', ERR=999)
```

If an error is detected during execution of this OPEN statement, control transfers to statement 999.

18.3 Using the IOSTAT Specifier

You can use the IOSTAT specifier to continue program execution after an I/O error and to return information about I/O operations. It can supplement or replace the END and ERR transfers. Execution of an I/O statement containing the IOSTAT specifier suppresses printing of an error message and causes the specified integer scalar memory reference to become defined as one of the following:

- A value of -1 if an end-of-file condition occurs
- A value of 0 if neither an error condition nor an end-of-file condition occurs
- A positive integer value if an error condition occurs (this value is one of the FORTRAN-specific error numbers listed in Table 18-1)

Following execution of the I/O statement and assignment of an IOSTAT value, control transfers to the END or ERR statement label, if any. If there is no control transfer, normal execution continues.

You can include SYS\$LIBRARY:FORSYSDEF.TLB(\$FORIOSDEF) in your program to obtain symbolic definitions for the values of IOSTAT. The symbolic names in this file have a form similar to the FORTRAN condition symbols:

Condition symbol	IOSTAT value
FOR\$_error	FOR\$IOS_error

Note that the values of the IOSTAT symbols are not the same as the values of the condition symbols described in Table 18-1.

The following example uses the IOSTAT specifier and the FORIOSDEF module to detect and process an OPEN error.

```

      CHARACTER*40 FILN
      INCLUDE '($FORIOSDEF)'
100  ACCEPT *, FILN
      OPEN (UNIT=1, FILE=FILN, STATUS='OLD', IOSTAT=IERR, ERR=100)
      .
      .
      .
      (process the input file)
      .
      .
      .
100  IF (IERR .EQ. FOR$IOS_FILNOTFOU) THEN
      TYPE *, 'File:', FILN, 'Does not exist, enter new filename'
      ELSE IF (IERR .EQ. FOR$IOS_FILNAMSPE) THEN
      TYPE *, 'File:', FILN, 'Was bad, enter new filename'
      ELSE
      TYPE *, 'Unrecoverable error, code =', IERR
      STOP
      END IF
      GO TO 10
      END

```


Appendix A

Additional Language Elements

For the purpose of facilitating compatibility with other versions of FORTRAN, VAX FORTRAN includes the statements ENCODE, DECODE, DEFINE FILE, and FIND, and it offers alternative syntax for the PARAMETER statement and octal constants. These language elements are particularly useful in transporting older FORTRAN programs to VAX, but should be avoided in new FORTRAN programs for use on VAX systems and in new programs for which portability to other FORTRAN-77 implementations is important.

The ANSI FORTRAN-77 interpretation of the EXTERNAL statement is incompatible with the previous standard and with previous DIGITAL implementations of FORTRAN. Section A.6 describes the interpretation of the EXTERNAL statement that applies when the /NOF77 compiler command qualifier is used.

A.1 The ENCODE and DECODE Statements

The ENCODE and DECODE statements transfer data between variables or arrays in internal storage and translate that data from internal to character form, and vice versa, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential WRITE and READ statements.

The ENCODE and DECODE statements have the forms:

```
ENCODE (c,f,b[,IOSTAT=ios][,ERR=s]) [list]
DECODE (c,f,b[,IOSTAT=ios][,ERR=s]) [list]
```

where:

c

is an integer expression. In the ENCODE statement, c is the number of characters (bytes) to be translated to character form. In the DECODE statement, c is the number of characters to be translated to internal form.

f

is a format identifier. If more than one record is specified, an error occurs.

- b** is a scalar reference or array name reference. In the ENCODE statement, b receives the characters after translation to external form. In the DECODE statement, b contains the characters to be translated to internal form.
- ios** is an integer scalar memory reference that is defined as a positive integer if an error occurs, and as a zero if no error occurs.
- s** is the label of an executable statement.
- list** is an I/O list. In the ENCODE statement, the I/O list contains the data to be translated to character form. In the DECODE statement, the list receives the data after translation to internal form.

Considerations/Restrictions

- The ENCODE statement translates the list elements to character form according to the format specifier, and stores the characters in b, as does a WRITE statement. If fewer than c characters are transmitted, the remaining character positions are filled with spaces.
- The DECODE statement translates the character data in b to internal (binary) form according to the format specifier, and stores the elements in the list, as does a READ statement.
- If b is an array, its elements are processed in the order of subscript progression.
- The number of characters that the ENCODE or DECODE statement can process depends on the data type of b. For example, an INTEGER*2 array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array. A character variable or character array element can contain characters equal in number to its length. A character array can contain characters equal in number to the length of each element multiplied by the number of elements.
- The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

Examples

An example of the ENCODE and DECODE statements follows:

```

DIMENSION K(3)
CHARACTER*12 A,B
DATA A/'123456789012'/
DECODE (12,100,A) K
100 FORMAT (3I4)
ENCODE (12,100,B) K(3), K(2), K(1)

```

The `DECODE` statement translates the 12 characters in `A` to integer form (specified by statement 100) and stores them in array `K`, as follows:

```
K(1) = 1234
K(2) = 5678
K(3) = 9012
```

The `ENCODE` statement translates the values `K(3)`, `K(2)`, and `K(1)` to character form and stores the characters in the character variable `B` as follows:

```
B = '901256781234'
```

A.2 DEFINE FILE Statement

The `DEFINE FILE` statement establishes the size and structure of relative organization files and associates them with a logical unit number. The `OPEN` statement performs the same function, and its use is preferred.

The `DEFINE FILE` statement has the form:

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

where:

u

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

m

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

n

is an integer constant or variable that specifies the length of each record in 16-bit words (2 bytes).

U

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

asv

is an integer variable, called the associated variable of the file. At the end of each direct access I/O operation, the record number of the next higher numbered record in the file is assigned to `v`; `asv` must not be a dummy argument.

The `DEFINE FILE` statement specifies that a file containing `m` fixed-length records, each composed of `n` 16-bit words, exists (or is to exist) on the specified logical unit. The records in the file are numbered sequentially from 1 through `m`.

A `DEFINE FILE` statement must be executed before the first direct access I/O statement referring to the specified file, even though the `DEFINE FILE` statement does not itself open the file. The file is actually opened when the first direct access I/O statement for the unit is

executed. If this I/O statement is a `WRITE`, a new relative organization file is created. If it is a `READ` or `FIND`, an existing file is opened—unless, of course, the specified file does not exist, in which case an error occurs.

The `DEFINE FILE` statement also establishes the integer variable `asv` as the associated variable of a file. At the end of each direct access I/O operation, the FORTRAN I/O system places in `asv` the record number of the record immediately following the one just read or written. Because the associated variable always points to the next sequential record in the file (unless the associated variable is redefined by an assignment, input, or `FIND` statement), direct access I/O statements can perform sequential processing on the file. They do this by using the associated variable of the file as the record number specifier.

For example:

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

This statement specifies that logical unit 3 is to be connected to a file of 1000 fixed-length records; each record is forty-eight 16-bit words long. The records are numbered sequentially from 1 through 1000 and are unformatted. After each direct access I/O operation on this file, the integer variable `NREC` will contain the record number of the record immediately following the record just processed.

A.3 FIND Statement

A `FIND` statement is similar to a direct access `READ` statement with no I/O list and can result in the opening of an existing file. The `FIND` statement positions a relative organization file to a particular record and sets the associated variable of the file to that record number. No data transfer takes place. See the description of the `OPEN` statement's `ASSOCIATEVARIABLE` keyword or the `DEFINE FILE` statement for information about associate variables.

The `FIND` statement has the forms:

```
FIND (u r[,ERR=s][,IOSTAT=ios])  
FIND ([UNIT=]u,REC=r[,ERR=s][,IOSTAT=ios])
```

where:

- u**
is a logical unit number.
- r**
is the direct access record number.
- s**
is the label of the executable statement to which control is to be transferred if no error occurs.

ios

is an integer variable or integer array element that is defined as a positive integer if an error occurs, and as a zero if no error occurs.

The unit number must refer to a relative organization file.

The record number cannot be less than one or greater than the number of records defined for the file.

For example:

```
FIND (1'1)
```

This statement positions logical unit 1 to the first record of the file; the file's associated variable is set to one.

```
FIND (4'INDX)
```

This statement positions the file to the record identified by the content of `INDX`; the file's associated variable is set to the value of `INDX`.

A.4 PARAMETER Statement

The `PARAMETER` statement has two forms. Both forms of the `PARAMETER` statement assign a symbolic name to a constant. The `PARAMETER` statement discussed here differs from the `PARAMETER` statement discussed in Section 8.11 in the following ways: its list is not bounded with parentheses, and the form of the constant, rather than implicit or explicit typing of the symbolic name, determines the data type of the variable.

The `PARAMETER` statement has the form:

```
PARAMETER p=c [,p=c]...
```

where:

p

is a symbolic name.

c

is a constant, the symbolic name of a constant, or a compile-time constant expression.

Each symbolic name (`p`) becomes a constant and is defined as the value of the constant or constant expression (`c`). Once a symbolic name is defined as a constant, it can appear in any position in which a constant is allowed. The effect is the same as if the constant were written there instead of the symbolic name.

The symbolic name of a constant cannot appear as part of another constant, but it can appear as a real or imaginary part of a complex constant.

Compile-time constant expressions are defined in Section 8.11.

You can use a symbolic name in a PARAMETER statement only to identify the symbolic name's corresponding constant in that program unit. Such a name can be defined only once in PARAMETER statements within the same program unit.

The symbolic name of a constant assumes the data type of its corresponding constant expression. The initial letter of the constant's name does not affect its data type. You cannot specify the data type of a parameter constant in a type declaration statement.

For example:

```
PARAMETER PI=3.1415927, DPI=3.141592653589793238D0
PARAMETER PIOV2=PI/2, DPIOV2=DPI/2
PARAMETER FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS'
```

A.5 Octal Notation for Integer Constants

Octal forms of integer constants are provided for compatibility with PDP-11 FORTRAN.

The octal form of an integer constant is:

"nn

where:

nn

is a string of digits in the range 0 to 7.

Examples of valid and invalid octal integer constants are:

Valid	Invalid (with explanation)
"107	"108 (contains a digit outside the allowed range)
"177777	"1377. (contains a decimal point)
"17777"	"17777" (contains a trailing quotation mark)

Note that these octal forms are not the same as the typeless octal constants discussed in Section 6.2.1.4. Integer constants in octal form have integer data type and are treated as integers.

A.6 /NOF77 Interpretation of the EXTERNAL Statement

The /NOF77 interpretation of the EXTERNAL statement combines the function of the INTRINSIC statement with that of the EXTERNAL statement discussed in Section 8.7. It is available only if the /NOF77 compiler command qualifier is present.

The /NOF77 EXTERNAL statement allows the programmer to use subprograms as arguments to other subprograms. The subprograms to be used as arguments can be either user-supplied procedures or FORTRAN library functions.

The /NOF77 EXTERNAL statement has the form:

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

where:

v

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

specifies that a user-supplied function is to be used instead of a FORTRAN library function having the same name. See Section 10.3 for information on FORTRAN library functions (intrinsic functions).

The /NOF77 EXTERNAL statement declares that each symbolic name in its list is an external procedure name. Such a name can then be used as an actual argument to a subprogram, which in turn can use the corresponding dummy argument in a function reference or CALL statement.

Note however, that a complete function reference used as an argument represents a value, not a subprogram name, for example, SQRT(B) in CALL SUBR(A, SQRT(B), C). It is not, therefore, defined in an EXTERNAL statement (as would be the incomplete reference SQRT).

An example of the /NOF77 EXTERNAL statement follows:

Main Program

```
EXTERNAL SIN, COS, *TAN, SINDEG
  ,
  ,
CALL TRIG(ANGLE,SIN,SINE)
  ,
  ,
CALL TRIG(ANGLE,COS,COSINE)
  ,
  ,
CALL TRIG(ANGLE,TAN,TANGNT)
  ,
  ,
CALL TRIG(ANGLED,SINDEG,SINE)
  ,
  ,
  ,
```

Subprograms

```
SUBROUTINE TRIG(X,F,Y)
  Y = F(X)
  RETURN
END

FUNCTION TAN(X)
  TAN = SIN(X)/COS(X)
  RETURN
END

FUNCITON SINDEG(X)
  SINDEG = SIN(X)*3.14159/80)
  RETURN
END
```

The CALL statements pass the name of a function to the subroutine TRIG. The function reference F(X) subsequently invokes the function in the second statement of TRIG. Depending on which CALL statement invoked TRIG, the second statement is equivalent to one of the following:

Y = SIN(X)
Y = COS(X)
Y = TAN(X)
Y = SINDEG(X)

The functions SIN and COS are examples of trigonometric functions supplied in the FORTRAN library. The function TAN is also supplied in the library. But the asterisk in the EXTERNAL statement specifies that the user-supplied function be used, instead of the library function. The function SINDEG is also a user-supplied function. Because no library function has the same name, no asterisk is required.

Appendix B

Character Sets

B.1 FORTRAN Character Set

The FORTRAN character set consists of the following:

- All upper- and lowercase letters (A through Z, a through z)
- The numerals 0 through 9
- The following special characters:

Character	Name	Character	Name
△ or TAB	Space or tab	'	Apostrophe
=	Equal sign	"	Quotation mark
+	Plus sign	\$	Dollar sign
-	Minus sign	—	Underscore
*	Asterisk	!	Exclamation point
/	Slash	:	Colon
(Left parenthesis	<	Left angle bracket
)	Right parenthesis	>	Right angle bracket
,	Comma	%	Percent sign
.	Period	&	Ampersand

Other printing characters can appear in a FORTRAN statement only as part of a Hollerith or character constant. Any printing character can appear in a comment. Printing characters are characters whose ASCII codes are in the range 20 through 7D. See Table B-1.

B.2 ASCII Character Set

Table B-1 represents the ASCII character set. At the top of the table are hexadecimal digits (0 to 7), and to the left of the table are hexadecimal digits (0 to F). To determine the hexadecimal value of an ASCII character, use the hexadecimal digit that corresponds to the row in the “units” position, and use the hexadecimal digit that corresponds to the column in the “16’s” position. For example, the value of the character representing the equal sign is 3D.

Table B-1: ASCII Character Set

		Column							
		0	1	2	3	4	5	6	7
Row	0	NUL	DLE	SP	0	@	P		p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	=	2	B	R	b	r
	3	ETX	DC3	△	3	C	S	C	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	BEL	ETB	'	7	G	W	g	w
	8	BS	CAN	(8	H	X	h	x
	9	HT	EM)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[k	{
	C	FF	FS	,	<	L	\	l	
	D	CR	GS	-	=	M]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation	EM	End of Medium
LF	Line Feed	SUB	Substitute
VT	Vertical Tab	ESC	Escape
FF	Form Feed	FS	File Separator
CR	Carriage Return	GS	Group Separator
SO	Shift Out	RS	Record Separator
SI	Shift In	US	Unit Separator
SP	Space	DEL	Delete

B.3 Radix-50 Constants and Character Set

Radix-50 is a special character data representation in which up to 3 characters can be encoded and packed into 16 bits. The Radix-50 character set is a subset of the ASCII character set and is provided for compatibility with PDP-11 FORTRAN.

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

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

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

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

where:

i, j, and k

represent the code values of three Radix-50 characters.

Thus, the maximum Radix-50 value is:

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

A Radix-50 constant has the form:

$$nRc_1c_2...c_n$$

where:

n

is an unsigned, nonzero integer constant that states the number of characters to follow.

c

is a character from the Radix-50 character set.

The maximum number of characters is 12. The character count must include any spaces that appear in the character string (the space character is a valid Radix-50 character). You can use Radix-50 constants only in DATA statements.

Examples of valid and invalid Radix-50 constants are:

Valid	Invalid (with explanation)
4RABCD	4RDK0: (colon is not a Radix-50 character)
6R△TO△△△	

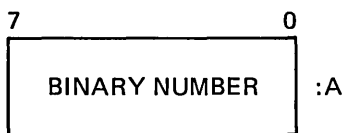
When a Radix-50 constant is assigned to a numeric variable or array element, the number of bytes that can be assigned depends on the data type of the component (refer to Table 6-1). If the Radix-50 constant contains fewer bytes than the length of the component, ASCII null characters (zero bytes) are appended on the right. If the constant contains more bytes than the length of the component, the rightmost characters are not used.

Appendix C

FORTRAN Data Representation

This appendix describes the data types supported by VAX FORTRAN and illustrates how they are stored in memory. The symbol :A in any illustration specifies the address of the byte containing bit 0, the starting address of the data element represented.

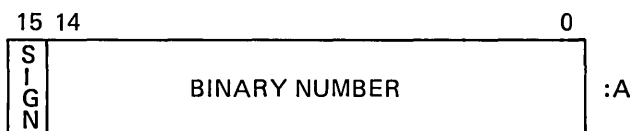
C.1 LOGICAL*1 (BYTE) Representation



ZK-797-82

LOGICAL*1 (or BYTE) values are in the range -128 to 127.

C.2 INTEGER*2 Representation



ZK-798-82

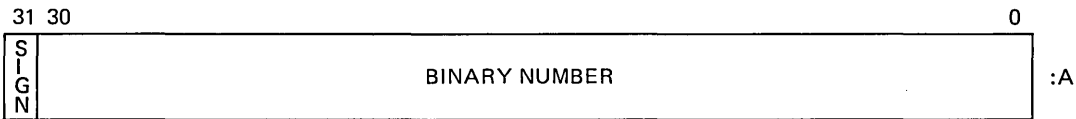
SIGN = 0(+), 1(-)

Integers are stored in a two's complement representation. INTEGER*2 values are in the range -32768 to 32767, and are stored in two contiguous bytes aligned on an arbitrary byte boundary. For example:

+22 = 0016(hex)

-7 = FFF9(hex)

C.3 INTEGER*4 Representation



ZK-799-82

SIGN = 0(+), 1(-)

INTEGER*4 values are stored in twos complement representation and lie in the range -2147483648 to 2147483647. Each value is stored in four contiguous bytes, aligned on an arbitrary byte boundary. Note that if the value is in the range of an INTEGER*2 value, that is, -32768 to 32767, then the first word can be referenced as an INTEGER*2 value.

C.4 Floating-Point Representations

The exponent for the REAL*4 and REAL*8 (D__floating) formats is stored in binary excess 128 notation. Binary exponents from -127 to 127 are represented by the binary equivalents of 1 through 255.

The exponent for the REAL*8 (G__floating) format is stored in binary excess 1024 notation. The exponent for the REAL*16 format is stored in binary excess 16384 notation. In REAL*8 (G__floating) format, binary exponents from -1023 to 1023 are represented by the binary equivalents of 1 through 2047. In REAL*16 format, binary exponents from -16383 to 16383 are represented by the binary equivalents of 1 through 32767.

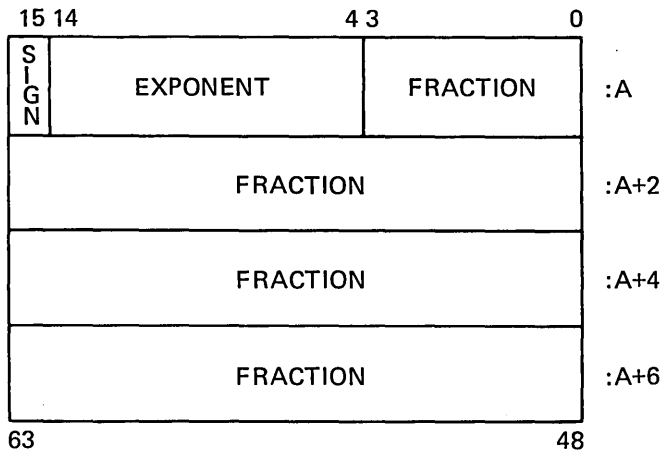
For each floating-point format, fractions are represented in sign-magnitude notation, with the binary radix point to the left of the most significant bit. Fractions are assumed to be normalized, and therefore the most significant bit is not stored (this is called "hidden bit normalization"). This bit is assumed to be 1 unless the exponent is 0. If the exponent equals 0, then the value represented is either zero (refer to the section entitled "Representation of 0.0" in *VAX FORTRAN User's Guide*) or is a reserved operand (refer to the section entitled "Reserved Operand Faults" in *VAX FORTRAN User's Guide*).

C.4.1 REAL*4 (F__floating)

REAL*4 (F__floating) data is four contiguous bytes starting on an arbitrary byte boundary. Bits are labeled from the right, 0 through 31.

C.4.3 REAL*8 (G_floating)

REAL*8 (G_floating) data is eight contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right, 0 through 63.



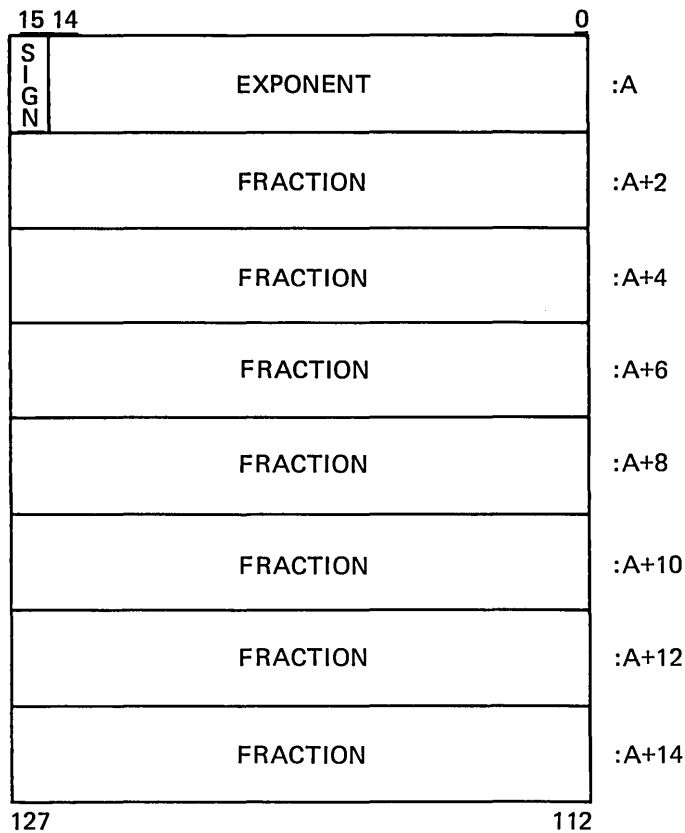
ZK-804-82

SIGN = 0(+), 1(-)

The form of REAL*8 (G_floating) data is sign magnitude, with bit 15 the sign bit, bits 14:4 an excess 1024 binary exponent, and bits 3:0 and 63:16 a normalized 53-bit fraction with the redundant most significant fraction bit not represented. The value of a G_floating data is in the approximate range 0.56×10^{-308} through 0.9×10^{308} . The precision of G_floating data is approximately one part in 2^{52} , that is, typically 15 decimal digits.

C.4.4 REAL*16 (H_floating)

REAL*16 (H_floating) data is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right, 0 through 127.



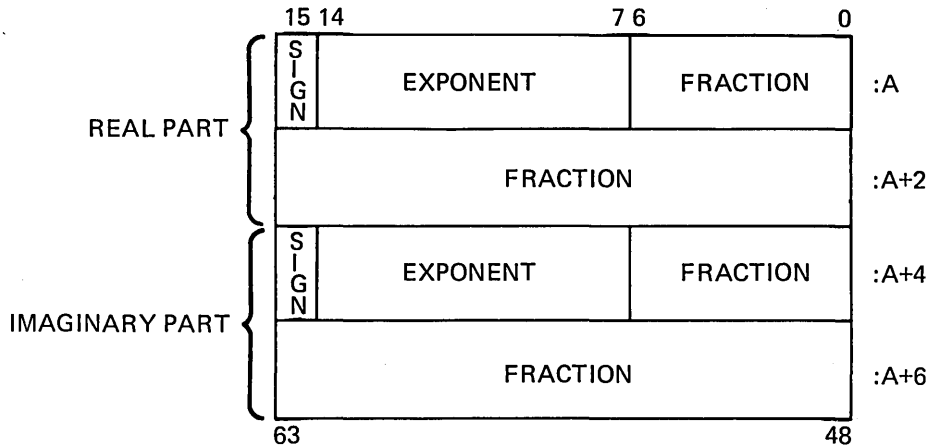
ZK-805-82

SIGN = 0(+), 1(-)

The form of a REAL*16 (H_floating) data is sign magnitude with bit 15 the sign bit, bits 14:0 an excess 16384 binary exponent, and bits 127:16 a normalized 113-bit fraction with the redundant most significant fraction bit not represented. The value of H_floating data is in the approximate range $0.84 \cdot 10^{-4932}$ through $0.59 \cdot 10^{4932}$. The precision of H_floating data is approximately one part in 2^{112} , that is, typically 33 decimal digits.

C.4.5 COMPLEX*8 (F__floating)

COMPLEX*8 data is eight contiguous bytes aligned on an arbitrary byte boundary. The low-order four bytes contain REAL*4 data that represents the real part of the complex number. The high-order four bytes contain REAL*4 data that represents the imaginary part of the complex number.

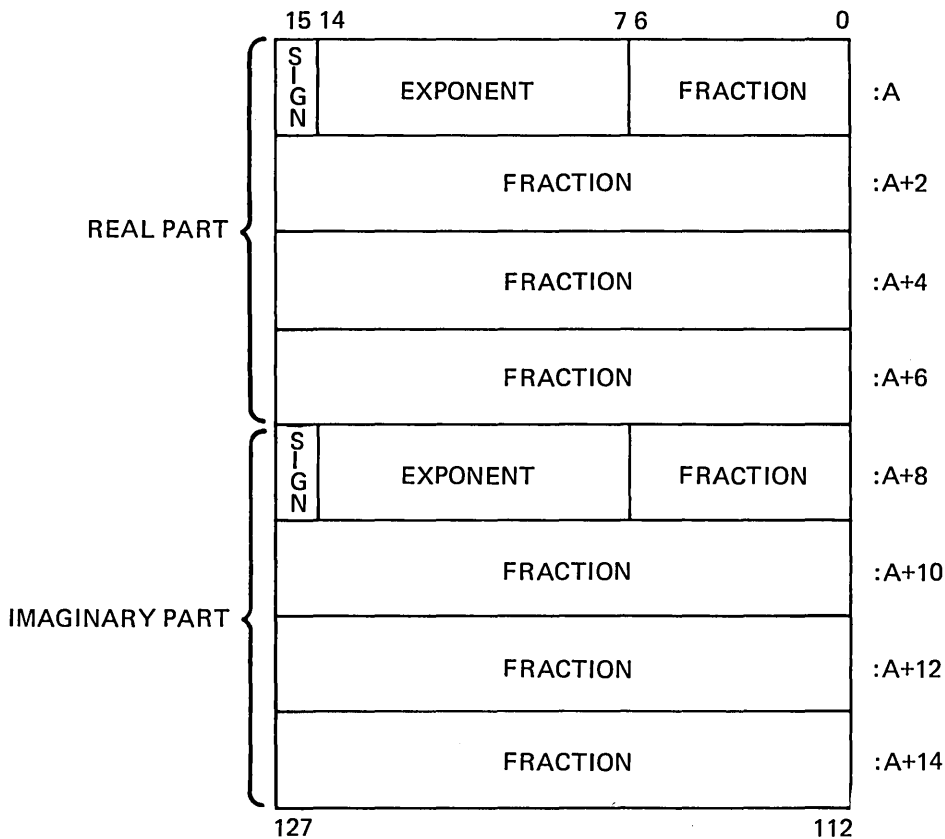


ZK-806-82

SIGN = 0(+), 1(-)

C.4.6 COMPLEX*16 (D__floating)

COMPLEX*16 (D__floating) data is 16 contiguous bytes aligned on an arbitrary byte boundary. The low-order eight bytes contain REAL*8 (D__floating) data that represents the real part of the complex data. The high-order eight bytes contain REAL*8 (D__floating) data that represents the imaginary part of the complex data.

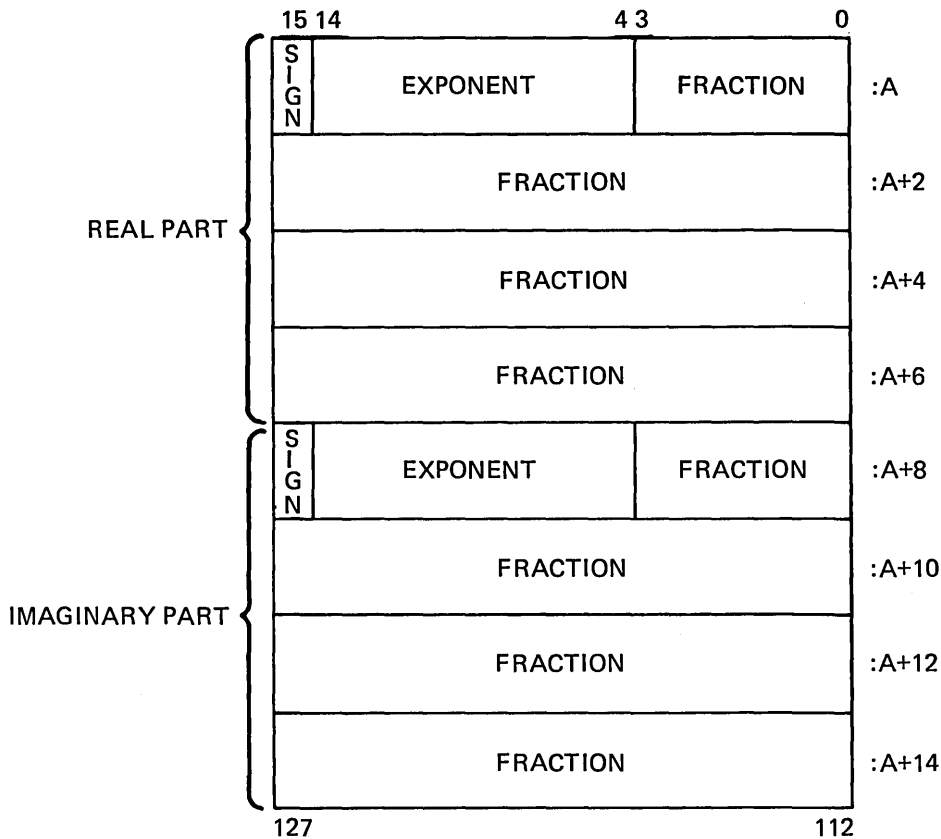


ZK-807-82

SIGN = 0(+), 1(-)

C.4.7 COMPLEX*16 (G__floating)

COMPLEX*16 (G__floating) data is 16 contiguous bytes aligned on an arbitrary byte boundary. The low-order eight bytes contain REAL*8 (G__floating) data that represents the real part of the complex data. The high-order eight bytes contain REAL*8 (G__floating) data that represents the imaginary part of the complex data.

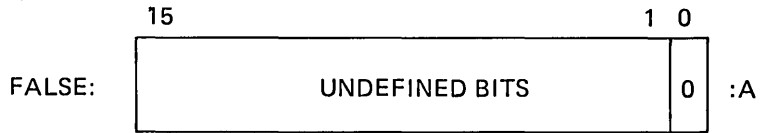
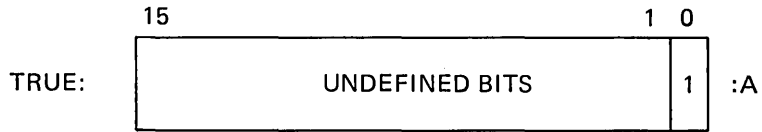


ZK-808-82

C.5 Logical Representation

Logical values are stored in two or four contiguous bytes, starting on an arbitrary byte boundary. The low-order bit (bit 0) determines the value. If bit 0 is set, the value is `.TRUE.`. If bit 0 is clear, the value is `.FALSE.`.

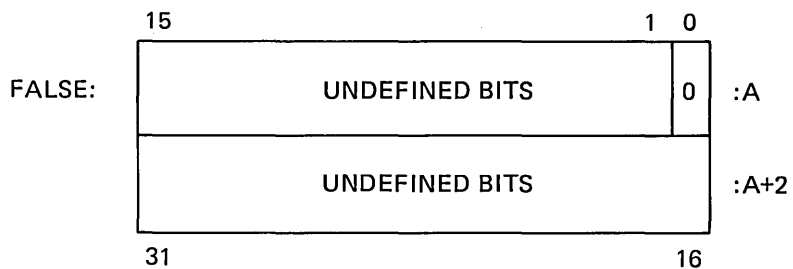
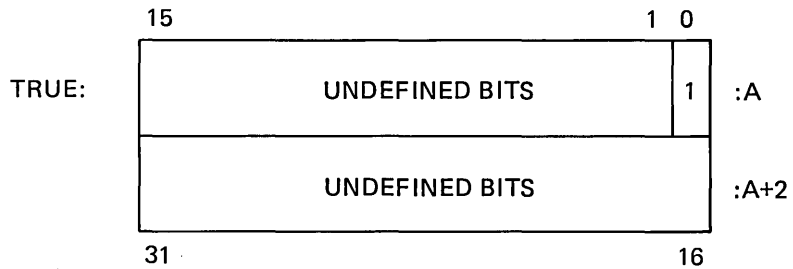
LOGICAL*2



Insert FA-8

ZK-802-82

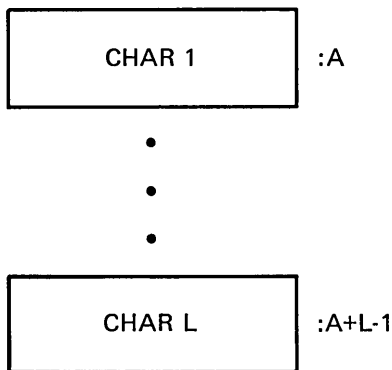
LOGICAL*4



ZK-803-82

C.6 Character Representation

A character string is a contiguous sequence of bytes in memory.



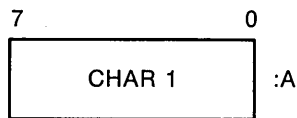
ZK-809-82

A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. The length L of a string is in the range 1 through 65535.

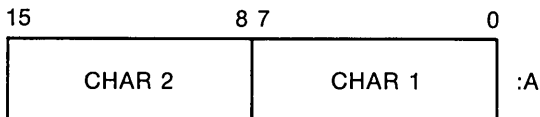
C.7 Hollerith Representation

Hollerith constants are stored internally, one character per byte.

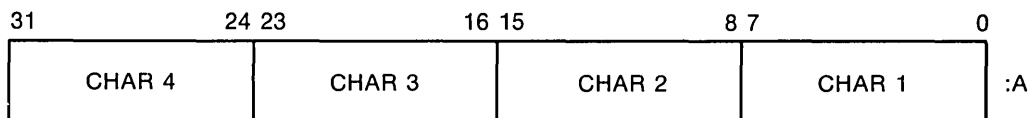
1 Byte



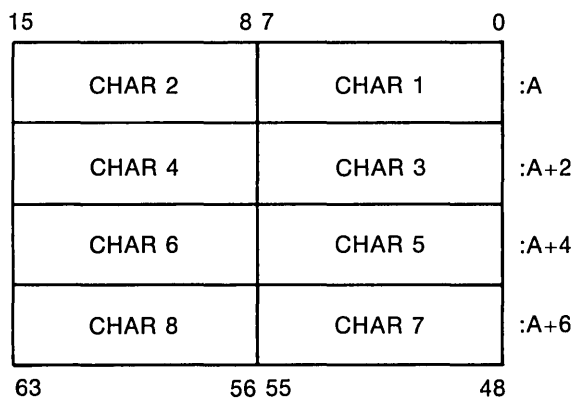
2 Bytes



4 Bytes



8 Bytes



ZK-810-82

Appendix D

FORTRAN Language Summary

D.1 Expression Operators

This section lists the expression operators in each data type in order of descending precedence:

Table D-1: Expression Operators

Data Type	Operator	Operation	Operates Upon
Arithmetic	**	Exponentiation	Arithmetic or logical expressions
	*,/	Multiplication, division	
	+,-	Addition, subtraction, unary plus and minus	
Character	//	Concatenation	Character expressions
Relational	.GT.	Greater than	Arithmetic, logical, or character expressions (all relational operators have equal precedence)
	.GE.	Greater than or equal to	
	.LT.	Less than	
	.LE.	Less than or equal to	
	.EQ.	Equal to	
	.NE.	Not equal to	

Table D-1 (Cont.): Expression Operators

Data Type	Operator	Operation	Operates Upon	
Logical	.NOT.	.NOT.A is true only if A is false	Logical or integer expressions	
	.AND.	A.AND.B is true only if A and B are both true		
	.OR.	A.OR.B is true if either A or B or both are true		
	.EQV.	A.EQV.B is true only if A and B are both true or A and B are both false		.EQV., .NEQV., and .XOR. have equal priority
	.NEQV.	A.NEQV.B is true only if A is true and B is false or B is true and A is false		
	.XOR.	Same as .NEQV.		

D.2 Statements

This section summarizes the statements available in the VAX FORTRAN language, including the general form of each statement. The statements are listed alphabetically for ease of reference. The "Manual Section" column indicates the section of this manual that describes each statement in detail.

Table D-2: VAX FORTRAN Statements

Statement Form	Description	Manual Section
ACCEPT	See READ.	11.7
ASSIGN <i>s</i> TO <i>v</i>		7.4
<i>s</i>	is the label of an executable statement or a FORMAT statement.	
<i>v</i>	is an integer variable name.	
Associates the statement label <i>s</i> with the integer variable <i>v</i> for later use as a format specifier or in an assigned GO TO statement.		

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
Assignment Statement v = e		7.1-7.3
v	is a scalar memory reference or an aggregate reference.	
e	is an expression or an aggregate.	
	The assignment statement assigns the value of the arithmetic, logical, or character expression on the right of the equal sign to the corresponding numeric, logical, or character scalar memory reference on the left. If aggregates are involved, the aggregate reference and the aggregate must have matching structures.	
BACKSPACE ([UNIT= <i>u</i>],[ERR= <i>s</i>],[IOSTAT= <i>ios</i>]) BACKSPACE <i>u</i>		13.5
u	is a logical unit specifier.	
s	is the label of an executable statement.	
ios	is an I/O status specifier.	
	The BACKSPACE statement backspaces the currently open file on logical unit <i>u</i> by one record.	
BLOCK DATA [<i>nam</i>]		8.1
nam	is a symbolic name.	
	The BLOCK DATA statement specifies the subprogram that follows as a BLOCK DATA subprogram.	
CALL sub([(a)],[a]...)]		9.1, 10.2.3
sub	is a subprogram name or entry point name.	
a	is an expression, an array name, a procedure name, or an alternate return specifier. An alternate return specifier is * <i>s</i> or & <i>s</i> , where <i>s</i> is the label of an executable statement.	
	The CALL statement calls the subroutine subprogram with the name specified by <i>s</i> , passing the actual arguments (<i>a</i>) to replace the dummy arguments in the subroutine definition.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
CLOSE ([UNIT= <i>u</i>],[<i>p</i>],[ERR= <i>s</i>],[IOSTAT= <i>ios</i>])		13.2
p	is one of the following parameters: $\left. \begin{array}{l} \text{STATUS} \\ \text{DISPOSE} \\ \text{DISP} \end{array} \right\} = \left(\begin{array}{l} \text{'SAVE'} \\ \text{'KEEP'} \\ \text{'DELETE'} \\ \text{'PRINT'} \\ \text{'SUBMIT'} \\ \text{'PRINT/DELETE'} \\ \text{'SUBMIT/DELETE'} \end{array} \right)$	
u	is a logical unit specifier.	
s	is the label of an executable statement.	
ios	is an I/O status specifier.	
	The CLOSE statement closes the specified file.	
COMMON <i>[/[cb]/nlist</i> [[, <i>]</i> <i>[/[cb]/nlist</i>] <i>...</i>		8.2
cb	is a common block name.	
nlist	is a list of one or more variable names, array names, array declarators, or records separated by commas.	
	The COMMON statement reserves one or more blocks of storage space to contain the variables associated with a specified block name.	
CONTINUE		9.2
	The CONTINUE statement transfers control to the next executable statement.	
DATA <i>nlist/clist</i> [[, <i>]</i> <i>nlist/clist</i>] <i>...</i>		8.4
nlist	is a list of one or more variable names, array names, array element names, character substring names, or implied-DO lists, separated by commas. Subscript expressions and substring expressions must be constant.	
clist	is a list of one or more constants separated by commas, each optionally preceded by <i>j</i> , where <i>j</i> is a non-zero, unsigned integer constant.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
	The DATA statement initially stores elements of clist in the corresponding elements of nlist.	
DECODE (c,f,b[,ERR=s][,IOSTAT=ios]) [list]		A.1
c	is an integer expression representing the number of characters to be translated to internal form.	
f	is a format identifier.	
b	is a scalar reference or array name reference that contains the characters to be translated to internal form.	
s	is the label of an executable statement.	
ios	is an integer scalar memory reference that is defined as a positive integer if an error occurs or as a zero if no error occurs.	
list	is an I/O list.	
	The DECODE statement reads c characters from buffer b and assigns values to the elements in the list converted according to format specification f.	
DEFINE FILE u(m,n,U,v)[,u(m,n,U,v)]...		A.2
u	is a logical unit specifier.	
m	specifies the number of records in the file.	
n	specifies the length of each record in 16-bit words.	
U	specifies unformatted.	
v	is an integer variable name.	
	The DEFINE FILE statement defines the record structure of a direct access file where u is the logical unit number, m is the number of fixed-length records in the file, n is the length in 16-bit words of a single record, U is a fixed argument, and v is the associated variable.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
DELETE ([UNIT= <i>u</i>],[REC= <i>r</i>],[ERR= <i>s</i>],[IOSTAT= <i>ios</i>])		13.7
DELETE (u 'r',[ERR= <i>s</i>],[IOSTAT= <i>ios</i>)		
u	is a logical unit specifier.	
r	is a record specifier.	
s	is the label of an executable statement.	
ios	is an I/O status specifier.	
<p>The DELETE statement deletes records from relative or indexed files, where <i>u</i> is the logical unit connected to the file, <i>r</i> is the number of the record in a relative file, <i>ios</i> is an I/O status specifier, and <i>s</i> is the label of the statement to which control is to be transferred if an error occurs.</p>		
DICTIONARY 'cdd-path/[NO]LIST'		3.5.3
cdd-path	is the full or relative pathname of a CDD object.	
[NO]LIST	directs the compiler to include or not include the generated FORTRAN source code in the listing.	
<p>The DICTIONARY statement extracts a data definition from the VAX Common Data Dictionary, translates it to FORTRAN source code, and includes it in a FORTRAN source program.</p>		
DIMENSION a(d)[,a(d)]...		8.5
a(d)	is an array declarator.	
a	is an array name.	
dl:du	are the lower (optional) and upper bounds of the array.	
<p>The DIMENSION statement specifies storage space requirements for arrays.</p>		

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
DO [s[,]] v=e1,e2[,e3]		9.3
s	is the label of an executable statement. VAX FORTRAN allows the statement label to be omitted.	
v	is a variable name.	
e1	is a numeric expression that specifies the initial value of v.	
e2	is a numeric expression that specifies the terminal value of the control variable.	
e3	is a numeric expression that specifies the value by which to increment the control variable.	
<p>The DO statement executes the DO loop by performing the following steps:</p> <ol style="list-style-type: none"> 1. Evaluates $cnt = INT((e2-e1+e3)/e3)$. 2. Sets $v = e1$. 3. If cnt is less than or equal to zero, does not execute the loop. 4. If cnt is greater than zero, then: <ol style="list-style-type: none"> a. Executes the statements in the body of the loop. b. Evaluates $v = v + e3$. c. Decrements the loop count ($cnt = cnt-1$). If cnt is greater than zero, repeats the loop. 		
DO [s[,]] WHILE (e)		9.3.2
s	is the label of an executable statement. VAX FORTRAN allows the statement label to be omitted.	
e	is a logical expression.	
<p>The DO WHILE statement is similar to the DO statement, but executes as long as the logical expression contained in the statement continues to be true, instead of for a specified number of iterations.</p>		

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
ELSE		9.7.3
	The ELSE statement defines a block of statements to be executed if logical expressions in previous IF THEN and ELSE IF THEN statements have values of false. See IF THEN .	
ELSE IF (e) THEN		9.7.3
e	is a logical expression.	
	The ELSE IF THEN statement defines a block of statements to be executed if logical expressions in previous IF THEN and ELSE IF THEN statements have values of false, and the logical expression e has a value of true. See IF THEN .	
ENCODE (c,f,b[,ERR=s][,IOSTAT=ios]) [list]		A.1
c	is an integer expression representing the number of characters (bytes) to be translated to character form.	
f	is a format identifier.	
b	is a scalar reference or array name reference.	
s	is a label of an executable statement.	
ios	is an integer scalar memory reference that is defined as a positive integer if an error occurs or as a zero if no error occurs.	
list	is an I/O list.	
	The ENCODE statement writes c characters into buffer b , which contains the values of the elements of the list, converted according to format specification f .	
END		9.5
	The END statement marks the end of a program unit.	
END DO		9.4
	The END DO statement marks the end of the body of a DO loop, and may be used in place of a labeled statement.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
END IF		9.7.3
	The END IF statement marks the end of a block IF construct.	
END MAP		8.15.3
	The END MAP statement marks the end of a field declaration or a series of field declarations.	
END STRUCTURE		8.15.1
	The END STRUCTURE statement marks the end of a structure declaration.	
END UNION		8.15.3
	The END UNION statement marks the end of a union declaration.	
ENDFILE ([UNIT= <i>u</i>],[ERR= <i>s</i>],[IOSTAT= <i>ios</i>]) ENDFILE <i>u</i>		13.6
	u is a logical unit specifier.	
	s is a label of an executable statement.	
	ios is an I/O status specifier.	
	The ENDFILE statement writes an end-of-file record on logical unit <i>u</i> .	
ENTRY <i>nam</i> ([(p[,p]...)])		10.2.4
	nam is a subprogram name.	
	p is a dummy argument or an alternate return specifier (*).	
	The ENTRY statement defines an alternate entry point within a subroutine or function subprogram.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
EQUIVALENCE (nlist)[,(nlist)]...		8.6
nlist	is a list of two or more variable names, array names, array element names, or character substring names separated by commas. Subscript expressions and substring expressions must be compile-time constant expressions. Records and record fields cannot be specified in EQUIVALENCE statements.	
	The EQUIVALENCE statement assigns the same storage location to each of the names in nlist.	
EXTERNAL v[,v]...		8.7, A.6
EXTERNAL *v[,*v]...		8.7
v	is a subprogram name.	
*	is used only if /NOF77 is specified.	
	The EXTERNAL statement defines the names specified as user-defined subprograms.	
FIND ((UNIT= <i>u</i> ,REC= <i>r</i> [,ERR= <i>s</i>][,IOSTAT= <i>ios</i>]) FIND (<i>u</i> ' <i>r</i> [,ERR= <i>s</i>][,IOSTAT= <i>ios</i>])		A.3
u	is a logical unit specifier.	
r	is a direct access record number.	
u 'r	is a logical unit specifier, not prefaced by UNIT=.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
	The FIND statement positions the file on logical unit <i>u</i> to record <i>r</i> and sets the associated variable to record number <i>r</i> .	
FORMAT (field-specification[,...])		12.1-12.8
	The FORMAT statement describes the format in which one or more records are to be transmitted; a statement label must be present.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
[typ] FUNCTION nam[*m]([(p[,p]...)])		10.2.2
typ	is a data type specifier.	
nam	is a function name.	
*m	is a data type length specifier.	
p	is a dummy argument.	
	The FUNCTION statement begins a function subprogram, indicating the program name and any dummy argument names (p). An optional type specification can be included.	
GO TO s		9.6.1
s	is a label of an executable statement.	
	This GO TO statement transfers control to statement number s.	
GO TO (slist)[,] e		9.6.2
slist	is a list of one or more statement labels separated by commas.	
e	is an integer expression.	
	This GO TO statement transfers control to the statement specified by the value of e (if e=1, control transfers to the first statement label; if e=2, control transfers to the second statement label, and so forth). If e is less than one or greater than the number of statement labels present, no transfer takes place.	
GO TO v[(,)(slist)]		9.6.3
v	is an integer variable name.	
slist	is a list of one or more statement labels separated by commas.	
	This GO TO statement transfers control to the statement most recently associated with v by an ASSIGN statement.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
IF (e) s1,s2,s3		9.7.1
e	is an expression.	
s1,s2,s3	are labels of executable statements.	
	This IF statement transfers control to statement s1, s2, or s3 depending on the value of e (if e is less than zero, control transfers to s1; if e equals zero, control transfers to s2; if e is greater than zero, control transfers to s3).	
IF (e) st		9.7.2
e	is an expression.	
st	is any executable statement except a DO, END DO, END, block IF, or logical IF.	
	This IF statement executes the statement if the logical expression has a value of true.	
IF (e1) THEN		9.7.3
block		
ELSE IF (e2) THEN		
block		
ELSE		
block		
END IF		
e1,e2	are logical expressions.	
block	is a series of zero or more FORTRAN statements.	
	This IF statement defines blocks of statements and conditionally executes them. If the logical expression in the IF THEN statement has a value of true, the first block is executed and control transfers to the first executable statement after the END IF statement.	
	If the logical expression has a value of false, the process is repeated for the next ELSE IF THEN statement. If all logical expressions have values of false, the ELSE block is executed. If there is no ELSE block, control transfers to the next executable statement following END IF.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
IMPLICIT typ(a[,a]...)[,typ(a[,a]...)]... IMPLICIT NONE		8.8
typ	is a data type specifier.	
a	is either a single letter, or two letters in alphabetical order, separated by a hyphen (that is, X-Y).	
NONE	inhibits the implicit declaration of variables in the module.	
<p>The IMPLICIT statement implicitly declares the data types of variables within program units. The element a represents a single (or a range of) letter(s) whose presence as the initial letter of a variable specifies the variable to be of that data type.</p> <p>IMPLICIT NONE and IMPLICIT must not be used in the same program unit.</p>		
INCLUDE 'file-spec'[/[NO]LIST]` INCLUDE '[file-spec](module-name)'[/[NO]LIST]`		3.5.1
file-spec	is a character constant that specifies the file to be included.	
module-name	is the name of a text module located in a text library.	
/[NO]LIST	indicates that the statements in the specified file are to be in the source listing.	
<p>The INCLUDE statement includes the source statements in the compilation from the file or module specified.</p>		
INQUIRE(par[,par]...)		13.3
par	is a keyword specification having the form: key = value	
where:		
key	is a keyword as described below.	
value	depends on the keyword.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
	<u>Keyword</u>	<u>Values</u>
	<u>inputs</u>	
	FILE	fin
	UNIT	e
	DEFAULTFILE	fin
	<u>outputs</u>	
	ACCESS	cv
	BLANK	cv
	CARRIAGECONTROL	cv
	DIRECT	cv
	ERR	s
	EXIST	lv
	FORM	cv
	FORMATTED	cv
	IOSTAT	v
	KEYED	cv
	NAME	cv
	NAMED	lv
	NEXTREC	v
	NUMBER	v
	OPENED	lv
	ORGANIZATION	cv
	RECL	v
	RECORDTYPE	cv
	SEQUENTIAL	cv
	UNFORMATTED	cv
e	is a numeric expression identifying a logical unit.	
fin	is a character expression identifying a file.	
v	is an integer scalar memory reference.	
lv	is a logical scalar memory reference.	
cv	is a character scalar memory reference.	
s	is a statement label.	
	The INQUIRE statement furnishes information on specified characteristics of a file or of a logical unit on which a file might be opened.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
INTRINSIC v[,v]...		8.9
v	is an intrinsic function name.	
	The INTRINSIC statement identifies symbolic names as representing intrinsic functions and allows those names to be used as actual arguments.	
Map Declaration (see Union Declaration)		
NAMelist /group-name/ namelist[[,] /group-name/ namelist]...		8.10
group-name	is a symbolic name.	
namelist	is a list of variables or array names, separated by commas, that is associated with the preceding group-name.	
	The NAMelist statement defines a list of variables or array names and associates that list of names with a unique group-name for use in namelist-directed I/O statements.	
OPEN (par[,par]...)		13.1
par	is a keyword specification in one of the following forms:	
	key	
	key = value	
key	is a keyword, as described below.	
value	depends on the keyword.	
	<u>Keyword</u>	<u>Values</u>
	ACCESS	'SEQUENTIAL' 'DIRECT' 'KEYED' 'APPEND'
	ASSOCIATEVARIABLE	v
	BLOCKSIZE	e
	BLANK	'NULL' 'ZERO'
	BUFFERCOUNT	e

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
	<u>Keyword</u>	<u>Values (Cont.)</u>
	CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'
	DEFAULTFILE	c
	DISP	(same as DISPOSE)
	DISPOSE	'KEEP' or 'SAVE' 'PRINT' 'DELETE' 'SUBMIT' 'SUBMIT/DELETE' 'PRINT/DELETE'
	ERR	s
	EXTENDSIZE	e
	FILE	c
	FORM	'FORMATTED' 'UNFORMATTED'
	INITIALSIZE	
	IOSTAT	v
	KEY	
	MAXREC	
	NAME	
	NOSPANBLOCKS	
	ORGANIZATION	
	READONLY	
	RECL	e
	RECORDSIZE	
	RECORDTYPE	
	SHARED	
	STATUS	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section									
	<table border="0"> <tr> <td><u>Keyword</u></td> <td><u>Values (Cont.)</u></td> </tr> <tr> <td>TYPE</td> <td>(same as STATUS)</td> </tr> <tr> <td>UNIT</td> <td>e</td> </tr> <tr> <td>USEROPEN</td> <td>p</td> </tr> </table>	<u>Keyword</u>	<u>Values (Cont.)</u>	TYPE	(same as STATUS)	UNIT	e	USEROPEN	p		
<u>Keyword</u>	<u>Values (Cont.)</u>										
TYPE	(same as STATUS)										
UNIT	e										
USEROPEN	p										
c	is a character scalar reference, numeric scalar memory reference, or numeric array name reference.										
e	is a numeric expression.										
p	is a program unit name.										
s	is a statement label.										
v	is an integer scalar memory reference.										
keyspec	is (e1:e2[:type]).										
where:											
e1	is the beginning byte of the key field.										
e2	is the ending byte of the key field.										
type	is either INTEGER or CHARACTER.										
	The OPEN statement opens a file on the specified logical unit according to the parameters specified by the keywords.										
OPTIONS qualifier[,qualifier...]		3.5.2									
qualifier	is one of the following:										
/NOCHECK											
/CHECK=	<table border="0"> <tr> <td>{</td> <td>ALL</td> <td>}</td> </tr> <tr> <td>{</td> <td>[(NO)OVERFLOW, [(NO)BOUNDS, [(NO)UNDERFLOW]</td> <td>}</td> </tr> <tr> <td>{</td> <td>NONE</td> <td>}</td> </tr> </table>	{	ALL	}	{	[(NO)OVERFLOW, [(NO)BOUNDS, [(NO)UNDERFLOW]	}	{	NONE	}	
{	ALL	}									
{	[(NO)OVERFLOW, [(NO)BOUNDS, [(NO)UNDERFLOW]	}									
{	NONE	}									
/[(NO)EXTENDSOURCE											
/[(NO)F77											
/[(NO)GFLOATING											
/[(NO)I4											
	The OPTIONS statement overrides the command line qualifiers for a single program unit.										

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
PARAMETER (p=c[,p=c]...)		8.11, A.4
p	is a symbolic name.	
c	is a constant, the name of a constant, or compile-time constant expression.	
The PARAMETER statement defines a symbolic name for a constant.		
PAUSE [disp]		9.8
disp	is a decimal digit string containing 1 to 5 digits or a character constant.	
The PAUSE statement displays a message on the screen and temporarily suspends program execution in order to permit you to take some action. You can respond by typing CONTINUE, EXIT, or DEBUG.		
PRINT	See WRITE.	11.8
PROGRAM nam		8.12
nam	is a program name.	
The PROGRAM statement specifies a name for the main program.		
READ Statement—Formatted Sequential Access		
READ ((UNIT=]u,[FMT=]f,[ERR=]s],[IOSTAT=ios],[END=]s) [list]		11.4.1.1
READ f[,list]		11.4.1.1
ACCEPT f[,list]		11.7
u	is a logical unit specifier.	
f	is the nonkeyword form of a format specifier.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
	This READ statement reads one or more logical records from unit <i>u</i> and assigns values to the elements in the list. The records are converted according to the format specifier (<i>f</i>).	
READ Statement—List-Directed Sequential Access		
READ ((UNIT= <i>u</i>],[FMT= <i>*</i>],[ERR= <i>s</i>],[IOSTAT= <i>ios</i>],[END= <i>s</i>]) [<i>list</i>]		11.4.1.2
READ <i>*</i> ,[<i>list</i>]		11.4.1.2
ACCEPT <i>*</i> ,[<i>list</i>]		11.7
u	is a logical unit specifier.	
*	denotes list-directed formatting.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	
	This READ statement reads one or more logical records from unit <i>u</i> and assigns values to the elements in the list. The records are converted according to the data type of the list element.	
READ Statement—Namelist-Directed Sequential Access		
READ ((UNIT= <i>u</i>],[NML= <i>nl</i>],[ERR= <i>s</i>],[IOSTAT= <i>ios</i>],[END= <i>s</i>])		11.4.1.3
READ <i>nl</i>		11.4.1.3
ACCEPT <i>n</i>		11.7
u	is a logical unit specifier.	
nl	is a namelist group-name.	
n	is the nonkeyword form of a namelist group-name specifier.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
	This READ statement reads one or more logical records from unit <i>u</i> and assigns values to specified namelist entities. The records are converted according to the data type of the namelist entities.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
READ Statement—Unformatted Sequential Access		
READ ([UNIT=]u[,ERR=s][,IOSTAT=ios][,END=s]) [list]		11.4.1.1
u	is a logical unit specifier.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	
This READ statement reads one unformatted record from unit u and assigns values to the elements in the list.		
READ Statement—Formatted Direct Access		
READ ([UNIT]=u[,FMT=]f[,REC=r[,ERR=s][,IOSTAT=ios]) [list]		11.4.2.1
READ (u 'r[,FMT=]f[,ERR=s][,IOSTAT=ios]) [list]		11.4.2.1
u	is a logical unit specifier.	
r	is a record specifier.	
u 'r	is a logical unit specifier, not prefaced by UNIT=.	
f	is a format specifier.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	
This READ statement reads record r from unit u and assigns values to the elements in the list. The record is converted according to f .		
READ Statement—Unformatted Direct Access		
READ ([UNIT=]u[,REC=r[,ERR=s][,IOSTAT=ios]) [list]		11.4.2.2
READ (u 'r[,ERR=s][,IOSTAT=ios]) [list]		11.4.2.2
u	is a logical unit specifier.	
r	is a record specifier.	
u 'r	is a logical unit specifier, not prefaced by UNIT=.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	
<p>This READ statement reads record r from unit u and assigns values to the elements in the list.</p>		
READ Statement—Formatted Indexed		
READ ([UNIT=]u,[FMT=]f,keyspec[,KEYID=kn][,ERR=s] [,IOSTAT=ios]) [list]		11.4.3.1
READ Statement—Unformatted Indexed		
READ ([UNIT=]u,keyspec[,KEYID=kn][,ERR=s] [,IOSTAT=ios]) [list]		11.4.3.2
u	is a logical unit specifier.	
f	is a format specifier.	
keyspec	is a key specifier (see Section 7.2.1.6).	
kn	is a key-of-reference specifier.	
s	is the label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	
<p>This READ statement reads one or more logical records specified by key value, and assigns values to the elements in the list.</p>		
READ Statement—Formatted Internal		
READ ([UNIT=]c,[FMT=]f[,ERR=s][,IOSTAT=ios][,END=s]) [list]		11.4.4
READ Statement—List-Directed Internal		
READ ([UNIT=]c,[FMT=]*[,ERR=s][,IOSTAT=ios][,END=s]) [list]		11.4.4
c	is an internal file specifier.	
*	denotes list-directed formatting.	
f	is a format specifier.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
	This READ statement reads into elements in the list one or more internal records containing character strings, converting in accordance with the format specification.	
RECORD structure-name/ record-namelist [,/structure-name/record-namelist] . . . [,/structure-name/record-namelist]	structure-name is the name of a previously declared structure. record-namelist is one or more variable names and/or array names. The RECORD statement creates a record for each variable specified or an array of records for each array specified. The structure declaration identified by structure-name defines the form of these records.	8.13
RETURN [i]	i is an integer value that indicates which alternate return is to be taken. The RETURN statement returns control to the calling program from the current subprogram.	9.9
REWIND ((UNIT=]u[,ERR=]s[,IOSTAT=ios) REWIND u	u is a logical unit specifier. s is a label of an executable statement. ios is an I/O status specifier. The REWIND statement repositions logical unit u to the beginning of the currently opened file.	13.4
REWRITE Statement—Formatted Indexed		
REWRITE ((UNIT=]u[,FMT=]f[,ERR=]s[,IOSTAT=ios) [list]		11.6.1

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
REWRITE Statement—Unformatted Indexed		
REWRITE ([UNIT= <i>u</i>],[ERR= <i>s</i>],[IOSTAT= <i>ios</i>]) [<i>list</i>]		11.6.2
u	is a logical unit specifier.	
f	is a format specifier.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	
The REWRITE statement transfers data from internal storage to the current record in an indexed file.		
SAVE [<i>a</i>],[<i>a</i>]...		8.14
a	is the name of a variable, an array, or a named common block enclosed in slashes.	
The SAVE statement retains the definition status of an entity after the execution of a RETURN or END statement in a subprogram.		
Statement Function		10.2.1
$f([p],[p]...)=e$		
f	is a statement function name.	
p	is a dummy argument.	
e	is an expression.	
A statement function creates a user-defined function having the variables <i>p</i> as dummy arguments. When referred to, the expression is evaluated using the actual arguments in the function call.		
STOP [<i>disp</i>]		9.10
disp	is a decimal digit string containing 1 to 5 digits or a character constant.	
The STOP statement terminates program execution and prints the display, if one is specified.		

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
Structure Declaration Block		
STRUCTURE	[/structure-name/] [field-namelist] field-declaration [field-declaration] . . . [field-declaration]	
END STRUCTURE		
structure-name	is the name that is used in RECORD statements to refer to a structure.	
field-namelist	are unique field names. (Used only in nested structure declarations.)	
field-declaration	is any declaration or combination of declarations of substructures, unions, or typed data.	
<p>A structure declaration block defines the field names, types of data within fields, and the order and alignment of fields within a record. Unlike type declaration statements, structure declarations do not create variables. Structured variables (called records) are created when you use a RECORD statement containing the name of a previously declared structure.</p>		
SUBROUTINE	nam([p[,p]...])	10.2.3
nam	is a subroutine name.	
p	is a dummy argument or an alternate return specifier ().	
<p>The SUBROUTINE statement begins a subroutine subprogram, indicating the program name and any dummy argument names (p).</p>		
TYPE	See WRITE.	11.8

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
Type Declarations—Character and Numeric		
Type Declaration (Character)		8.3.2
CHARACTER[*len[,]] v[*len] [/clist/] [,v[*len] [/clist/] ...		
len	specifies the length of the character data elements.	
v	is a variable name, array name, function or function entry name, or an array declarator. The name can optionally be followed by a data type length specifier (*n). For character entities, the length specifier can be *len or (*).	
clist	is an initial value or values to be assigned to the immediately preceding variable or array element.	
The character type declaration assigns the specified data type to the symbolic names (v).		
Type Declaration (Numeric)		5.3.1
typ v [/clist/] [,v [/clist/] ...		
typ	is any data type specifiers except CHARACTER, that is, BYTE, LOGICAL, LOGICAL*1, LOGICAL*2, LOGICAL*4, INTEGER, INTEGER*2, REAL, REAL*4, REAL*8, REAL*16, DOUBLE PRECISION, COMPLEX, COMPLEX*8, COMPLEX*16, DOUBLE COMPLEX.	
v	is a variable name, array name, function or function entry name, or an array declarator. The name can optionally be followed by a data type length specifier (*n).	
clist	is an initial value or values to be assigned to the immediately preceding variable or array element.	
The Numeric Type Declaration assigns the specified data type to the symbolic names (v).		

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
Union Declaration		8.15.3

UNION

```

map-declaration
map-declaration
[map-declaration]
.
.
.
[map-declaration]

```

END UNION

where **map-declaration** is:

```

MAP
  field-declaration
  [field-declaration]
  .
  .
  .
  [field-declaration]

```

END MAP

field-declaration

is any declaration or combination of declarations of substructures, unions, or typed data.

Unions define a data area that can be shared by fields or groups of fields at run time.

UNLOCK (([UNIT=]u[,ERR=s][,IOSTAT=ios])	13.8
---	------

UNLOCK u	13.8
----------	------

u is a logical unit specifier.

s is a label of an executable statement.

ios is an I/O status specifier.

The UNLOCK statement removes the access protection from the file connected to logical unit u.

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
VIRTUAL a(d)[,a(d)]...		8.5
a(d)	is an array declarator.	
a	is an array name.	
d	are the lower (optional) and upper bounds of the array in the form [dl:]du.	
	The VIRTUAL statement has the same effect as the DIMENSION statement and is included for compatibility with PDP-11 FORTRAN.	
VOLATILE nlist		8.16
nlist	is a list of one or more variable names, array names, or common block names separated by commas.	
	The VOLATILE statement prevents all optimizations for the items specified in the namelist.	
WRITE Statement—Formatted Sequential Access		
WRITE ([UNIT=]u,[FMT=]f[,ERR=s][,IOSTAT=ios]) [list]		11.5.1.1
PRINT f[,list]		11.8
TYPE f[,list]		11.8
u	is a logical unit specifier.	
f	is a format specifier.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	
	This WRITE statement writes one or more logical records to unit u containing the values of the elements in the list. The records are converted according to f.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
WRITE Statement—List-Directed Sequential Access		
WRITE ((UNIT=]u],[FMT=]*],[ERR=s],[IOSTAT=ios) [list]		11.5.1.2
PRINT f,[list]		11.8
TYPE f,[list]		11.8
u	is a logical unit specifier.	
*	denotes list-directed formatting.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	

This WRITE statement writes one or more logical records to unit *u* containing the values of the elements in the list. The records are converted according to the data type of the list element.

WRITE ((UNIT=]u],[NML=]nl],[ERR=s],[IOSTAT=ios)		11.5.1.3
PRINT n		11.8
TYPE n		11.8
u	is a logical unit specifier.	
n	is a nonkeyword form of a namelist group-name specifier.	
nl	is a namelist group-name.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	

This WRITE statement writes one or more logical records to unit *u* containing the values of the namelist entities. The records are converted according to the data type of the namelist entities.

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
WRITE Statement—Unformatted Sequential Access		
WRITE ((UNIT= <i>u</i> [,ERR= <i>s</i>][,IOSTAT= <i>ios</i>]) [<i>list</i>])		11.5.1.4
u	is a logical unit specifier.	
s	is a label of an executable statement label.	
ios	is an I/O status specifier.	
list	is an I/O list.	
This WRITE statement writes one unformatted record to unit <i>u</i> containing the values of the elements in the list.		
WRITE Statement—Formatted Direct Access		
WRITE ((UNIT= <i>u</i> [,FMT= <i>f</i> ,REC= <i>r</i> [,ERR= <i>s</i>][,IOSTAT= <i>ios</i>]) [<i>list</i>])		11.5.2.1
WRITE (<i>u</i> ´ <i>r</i> [, <i>f</i> [,ERR= <i>s</i>][,IOSTAT= <i>ios</i>]) [<i>list</i>])		11.5.2.1
u	is a logical unit specifier.	
r	is a record specifier.	
<i>u</i> ´ <i>r</i>	is a logical unit specifier, not prefaced by UNIT=.	
f	is a format specifier.	
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	
This WRITE statement writes the values of the elements of the list to record <i>r</i> on unit <i>u</i> . The record is converted according to <i>f</i> .		
WRITE Statement—Unformatted Direct Access		
WRITE ((UNIT= <i>u</i> [,REC= <i>r</i> [,ERR= <i>s</i>][,IOSTAT= <i>ios</i>]) [<i>list</i>])		11.5.2.2
WRITE (<i>u</i> ´ <i>r</i> [,ERR= <i>s</i>][,IOSTAT= <i>ios</i>]) [<i>list</i>])		11.5.2.2
u	is a logical unit specifier.	
r	is a record specifier.	
<i>u</i> ´ <i>r</i>	is a logical unit specifier, not prefaced by UNIT=.	

Table D-2 (Cont.): VAX FORTRAN Statements

Statement Form	Description	Manual Section
s	is a label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	
<p>This WRITE statement writes record r to unit u containing the values of the elements in the list.</p>		
WRITE Statement—Formatted Internal		
WRITE ([UNIT=]c,[FMT=]f[,ERR=s][,IOSTAT=ios]) [list]		11.5.4
WRITE Statement—List-Directed Internal		
WRITE ([UNIT=]c,[FMT=]*[,ERR=s][,IOSTAT=ios]) [list]		11.5.4
c	is an internal file specifier.	
*	denotes list-directed formatting.	
f	is a format specifier.	
s	is the label of an executable statement.	
ios	is an I/O status specifier.	
list	is an I/O list.	
<p>This WRITE statement writes elements in the list to the internal file specified by the unit, converting the elements to character strings in accordance with the format specification.</p>		

D.3 Library Functions

Table D-3 lists the VAX FORTRAN intrinsic functions. Superscripts in the table refer to the notes that follow the table. Refer to Section 10.3 for more information about intrinsic functions. For descriptions of the intrinsic function algorithms, refer to the *VAX/VMS Run-Time Library Routines Reference Manual*.

Table D-3: Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Square Root ¹ $a^{1/2}$	1	SQRT	SQRT	REAL*4	REAL*4
			DSQRT	REAL*8	REAL*8
			QSQRT	REAL*16	REAL*16
			CSQRT	COMPLEX*8	COMPLEX*8
			CDSQRT	COMPLEX*16	COMPLEX*16
Natural Logarithm ² $\log_e a$	1	LOG	ALOG	REAL*4	REAL*4
			DLOG	REAL*8	REAL*8
			QLOG	REAL*16	REAL*16
			CLOG	COMPLEX*8	COMPLEX*8
			CDLOG	COMPLEX*16	COMPLEX*16
Common Logarithm ² $\log_{10} a$	1	LOG10	ALOG10	REAL*4	REAL*4
			DLOG10	REAL*8	REAL*8
			QLOG10	REAL*16	REAL*16
Exponential e^a	1	EXP	EXP	REAL*4	REAL*4
			DEXP	REAL*8	REAL*8
			QEXP	REAL*16	REAL*16
			CEXP	COMPLEX*8	COMPLEX*8
			CDEXP	COMPLEX*16	COMPLEX*16
Sine ³ Sin a	1	SIN	SIN	REAL*4	REAL*4
			DSIN	REAL*8	REAL*8
			QSIN	REAL*16	REAL*16
			CSIN	COMPLEX*8	COMPLEX*8
			CDSIN	COMPLEX*16	COMPLEX*16
Sine ³ (degree) Sin a	1	SIND	SIND	REAL*4	REAL*4
			DSIND	REAL*8	REAL*8
			QSIND	REAL*16	REAL*16
Cosine ³ Cos a	1	COS	COS	REAL*4	REAL*4
			DCOS	REAL*8	REAL*8
			QCOS	REAL*16	REAL*16
			CCOS	COMPLEX*8	COMPLEX*8
			CDCOS	COMPLEX*16	COMPLEX*16
Cosine ³ (degree) Cos a	1	COSD	COSD	REAL*4	REAL*4
			DCOSD	REAL*8	REAL*8
			QCOSD	REAL*16	REAL*16
Tangent ³ Tan a	1	TAN	TAN	REAL*4	REAL*4
			DTAN	REAL*8	REAL*8
			QTAN	REAL*16	REAL*16
Tangent ³ (degree) Tan a	1	TAND	TAND	REAL*4	REAL*4
			DTAND	REAL*8	REAL*8
			QTAND	REAL*16	REAL*16
Arc Sine ^{4,5} Arc Sin a	1	ASIN	ASIN	REAL*4	REAL*4
			DASIN	REAL*8	REAL*8
			QASIN	REAL*16	REAL*16

Table D-3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Arc Sine (degree) Arc Sin a	1	ASIND	ASIND DASIND QASIND	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arc Cosine ^{4,5} Arc Cos a	1	ACOS	ACOS DACOS QACOS	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arc Cosine (degree) Arc Cos a	1	ACOSD	ACOSD DACOSD QACOSD	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arc Tangent ⁵ Arc Tan a	1	ATAN	ATAN DATAN QATAN	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arc Tangent ^{5,7} (degree) Arc Tan a	1	ATAND	ATAND DATAND QATAND	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arc Tangent ^{5,6} Arc Tan a ₁ /a ₂	2	ATAN2	ATAN2 DATAN2 QATAN2	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arc Tangent ^{5,7} (degree) Arc Tan a ₁ /a ₂	2	ATAN2D	ATAN2D DATAN2D QATAN2D	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Hyperbolic Sine Sinh a	1	SINH	SINH DSINH QSINH	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Hyperbolic Cosine Cosh a	1	COSH	COSH DCOSH QCOSH	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Hyperbolic Tangent Tanh a	1	TANH	TANH DTANH QTANH	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Absolute Value ⁸ a	1	ABS	IIABS JIABS ABS DABS QABS CABS CDABS	INTEGER*2 INTEGER*4 REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16	INTEGER*2 INTEGER*4 REAL*4 REAL*8 REAL*16 REAL*4 REAL*8
		IABS	IIABS JIABS	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
Truncation ^{9,12} a	1	INT	IINT JINT IIDINT JIDINT IIQINT	REAL*4 REAL*4 REAL*8 REAL*8 REAL*16	INTEGER*2 INTEGER*4 INTEGER*2 INTEGER*4 INTEGER*2

Table D-3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
			JIQINT	REAL*16	INTEGER*4
			—	COMPLEX*8	INTEGER*2
			—	COMPLEX*8	INTEGER*4
			—	COMPLEX*16	INTEGER*2
			—	COMPLEX*16	INTEGER*4
		IDINT	IIDINT	REAL*8	INTEGER*2
			JIDINT	REAL*8	INTEGER*4
		IQINT	IIQINT	REAL*16	INTEGER*2
			JIQINT	REAL*16	INTEGER*4
		AINT	AINT	REAL*4	REAL*4
			DINT	REAL*8	REAL*8
			QINT	REAL*16	REAL*16
Nearest Integer ^{9,12} [a + .5*sign(a)]	1	NINT	ININT	REAL*4	INTEGER*2
			JNINT	REAL*4	INTEGER*4
			IIDNNT	REAL*8	INTEGER*2
			JIDNNT	REAL*8	INTEGER*4
			IIQNNT	REAL*16	INTEGER*2
			JIQNNT	REAL*16	INTEGER*4
		IDNINT	IIDNNT	REAL*8	INTEGER*2
			JIDNNT	REAL*8	INTEGER*4
		IQNINT	IIQNNT	REAL*16	INTEGER*2
			JIQNNT	REAL*16	INTEGER*4
		ANINT	ANINT	REAL*4	REAL*4
			DNINT	REAL*8	REAL*8
			QNINT	REAL*16	REAL*16
Zero-Extend Functions	1	ZEXT	IZEXT	LOGICAL*1 LOGICAL*2	INTEGER*2
			JZEXT	LOGICAL*1 LOGICAL*2 LOGICAL*4 INTEGER*2 INTEGER*4	INTEGER*4
Conversion to ¹⁰ REAL*4	1	REAL	FLOATI	INTEGER*2	REAL*4
			FLOATJ	INTEGER*4	REAL*4
			—	REAL*4	REAL*4
			SNGL	REAL*8	REAL*4
			SNGLQ	REAL*16	REAL*4
			—	COMPLEX*8	REAL*4
			—	COMPLEX*16	REAL*4
Conversion to ¹⁰ REAL*8	1	DBLE	—	INTEGER*2	REAL*8
			—	INTEGER*4	REAL*8
			DBLE	REAL*4	REAL*8
			—	REAL*8	REAL*8
			DBLEQ	REAL*16	REAL*8
			—	COMPLEX*8	REAL*8
			—	COMPLEX*16	REAL*8

Table D-3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Conversion to REAL*16	1	QEXT	—	INTEGER*2	REAL*16
			—	INTEGER*4	REAL*16
			QEXT	REAL*4	REAL*16
			QEXTD	REAL*8	REAL*16
			—	REAL*16	REAL*16
			—	COMPLEX*8	REAL*16
			—	COMPLEX*16	REAL*16
Fix ^{10,12} (REAL*4-to-integer conversion)	1	IFIX	IIFIX	REAL*4	INTEGER*2
			JIFIX	REAL*4	INTEGER*4
Float ¹⁰ (Integer-to-REAL*4 conversion)	1	FLOAT	FLOATI	INTEGER*2	REAL*4
			FLOATJ	INTEGER*4	REAL*4
REAL*8 Float ¹⁰ (Integer-to-REAL*8 conversion)	1	DFLOAT	DFLOTI	INTEGER*2	REAL*8
			DFLOTJ	INTEGER*4	REAL*8
			QFLOAT	—	INTEGER*2 INTEGER*4
Conversion to COMPLEX*8, or COMPLEX*8 from Two Arguments	1,2 ¹³	CMPLX	—	INTEGER*2	COMPLEX*8
			—	INTEGER*4	COMPLEX*8
			—	REAL*4	COMPLEX*8
			—	REAL*8	COMPLEX*8
			—	REAL*16	COMPLEX*8
			—	COMPLEX*8	COMPLEX*8
			—	COMPLEX*16	COMPLEX*8
Conversion to COMPLEX*16, or COMPLEX*16 from Two Arguments	1,2 ¹³	DCMPLX	—	INTEGER*2	COMPLEX*16
			—	INTEGER*4	COMPLEX*16
			—	REAL*4	COMPLEX*16
			—	REAL*8	COMPLEX*16
			—	REAL*16	COMPLEX*16
			—	COMPLEX*8	COMPLEX*16
			—	COMPLEX*16	COMPLEX*16
Real Part of Complex	1	—	REAL	COMPLEX*8	REAL*4
			DREAL	COMPLEX*16	REAL*8
Imaginary Part of Complex	1	—	AIMAG	COMPLEX*8	REAL*4
			DIMAG	COMPLEX*16	REAL*8
Complex from Two Arguments	(See Conversion to COMPLEX*8 and Conversion to COMPLEX*16)				
Complex Conjugate (if a = (X, Y) CONJG (a) = (X, -Y))	1	CONJG	CONJG	COMPLEX*8	COMPLEX*8
			DCONJG	COMPLEX*16	COMPLEX*16
REAL*8 product of REAL*4's a ₁ *a ₂	2	—	DPROD	REAL*4	REAL*8

Table D-3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result		
Maximum¹² $\max(a_1, a_2, \dots, a_n)$ (returns the maximum value from among the argument list; there must be at least two arguments)	n	MAX	IMAX0	INTEGER*2	INTEGER*2		
			JMAX0	INTEGER*4	INTEGER*4		
			AMAX1	REAL*4	REAL*4		
			DMAX1	REAL*8	REAL*8		
			QMAX1	REAL*16	REAL*16		
		MAX0	IMAX0	INTEGER*2	INTEGER*2		
			JMAX0	INTEGER*4	INTEGER*4		
		MAX1	IMAX1	REAL*4	INTEGER*2		
			JMAX1	REAL*4	INTEGER*4		
		AMAX0	AIMAX0	INTEGER*2	REAL*4		
AJMAX0	INTEGER*4		REAL*4				
Minimum¹² $\min(a_1, a_2, \dots, a_n)$ (returns the minimum value among the argument list; there must be at least two arguments)	n	MIN	IMIN0	INTEGER*2	INTEGER*2		
			JMIN0	INTEGER*4	INTEGER*4		
			AMIN1	REAL*4	REAL*4		
			DMIN1	REAL*8	REAL*8		
			QMIN1	REAL*16	REAL*16		
		MIN0	IMIN0	INTEGER*2	INTEGER*2		
			JMIN0	INTEGER*4	INTEGER*4		
		MIN1	IMIN1	REAL*4	INTEGER*2		
			JMIN1	REAL*4	INTEGER*4		
		AMIN0	AIMIN0	INTEGER*2	REAL*4		
			AJMIN0	INTEGER*4	REAL*4		
		Positive Difference $a_1 - (\min(a_1, a_2))$ (returns the first argument minus the minimum of the two arguments)	2	DIM	IIDIM	INTEGER*2	INTEGER*2
					JIDIM	INTEGER*4	INTEGER*4
DIM	REAL*4				REAL*4		
DDIM	REAL*8				REAL*8		
QDIM	REAL*16				REAL*16		
IDIM	IIDIM			INTEGER*2	INTEGER*2		
	JIDIM			INTEGER*4	INTEGER*4		
Remainder $a_1 - a_2 * [a_1 / a_2]$ (returns the remainder when the first argument is divided by the second)	2	MOD	IMOD	INTEGER*2	INTEGER*2		
			JMOD	INTEGER*4	INTEGER*4		
			AMOD	REAL*4	REAL*4		
			DMOD	REAL*8	REAL*8		
			QMOD	REAL*16	REAL*16		
Transfer of Sign $ a_1 \text{ Sign } a_2$	2	SIGN	IISIGN	INTEGER*2	INTEGER*2		
			JISIGN	INTEGER*4	INTEGER*4		
			SIGN	REAL*4	REAL*4		
			DSIGN	REAL*8	REAL*8		
			QSIGN	REAL*16	REAL*16		
		ISIGN	IISIGN	INTEGER*2	INTEGER*2		
			JISIGN	INTEGER*4	INTEGER*4		
Bitwise AND (performs a logical AND on corresponding bits)	2	IAND	IIAND	INTEGER*2	INTEGER*2		
			JIAND	INTEGER*4	INTEGER*4		

Table D-3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Bitwise OR (performs an inclusive OR on corresponding bits)	2	IOR	IIOR JIOR	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
Bitwise Exclusive OR (performs an exclusive OR on corresponding bits)	2	IEOR	IIEOR JIEOR	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
Bitwise Complement (complements each bit)	1	NOT	INOT JNOT	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
Bitwise Shift (a_1 logically shifted left a_2 bits)	2	ISHFT	IISHFT JISHFT	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
Bit Extraction (extracts bits a_2 through $a_2 + a_3 - 1$ from a_1); see also MVBITS system subroutine	3	IBITS	IIBITS JIBITS	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
Bit Set (returns the value of a_1 with bit a_2 of a_1 set to 1)	2	IBSET	IIBSET JIBSET	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
Bit Test (returns .TRUE. if bit a_2 of argument a_1 equals 1)	2	BTEST	BITEST BJTEST	INTEGER*2 INTEGER*4	LOGICAL*2 LOGICAL*4
Bit Clear (returns the value of a_1 with bit a_2 of a_1 set to 0)	2	IBCLR	IIBCLR JIBCLR	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
Bitwise Circular Shift ¹⁴ (circularly shifts rightmost a_3 bits of argument a_1 by a_2 places)	3	ISHFTC	IISHFTC JISHFTC	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
Length ¹² (returns length of the character expression)	1	—	LEN	CHARACTER	INTEGER*4
Index (C_1, C_2) ¹² (returns the position of the substring c_2 in the character expression c_1)	2	—	INDEX	CHARACTER	INTEGER*4
Character ¹² (returns a character that has the ASCII value specified by the argument)	1	—	CHAR	LOGICAL*1 INTEGER*2 INTEGER*4	CHARACTER

Table D-3 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
ASCII Value ¹¹ (returns the ASCII value of the argument; the argument must be a character expression that has a length of 1)	1	—	ICHAR	CHARACTER	INTEGER*4
Character relationals (ASCII collating sequence)	2	—	LLT	CHARACTER	LOGICAL*4
	2	—	LLE	CHARACTER	LOGICAL*4
	2	—	LGT	CHARACTER	LOGICAL*4
	2	—	LGE	CHARACTER	LOGICAL*4

Notes

1. The argument of SQRT, DSQRT, or QSQRT must be greater than or equal to zero. The result of CSQRT or CDSQRT is the principal value, with the real part greater than or equal to zero. When the real part is zero, the result is the principal value, with the imaginary part greater than or equal to zero.
2. The argument of ALOG, DLOG, QSQRT, ALOG10, DLOG10, QLOG10, ATAND, ATAN2D, ASIND, DASIND, ACOSD, DACOSD, or QACOSD must be greater than zero. The argument of CLOG or CDLOG must not be (0.,0.).
3. The argument of SIN, DSIN, QSIN, COS, DCOS, QCOS, TAN, DTAN, or QTAN must be in radians. The argument is treated modulo 2π . The argument of SIND, COSD, or TAND must be in degrees. The argument is treated modulo 360.
4. The absolute value of the argument of ASIN, DASIN, QASIN, ACOS, DACOS, QACOS, ASIND, DASIND, QASIND, ACOSD, DACOSD, or QACOSD must be less than or equal to 1.
5. The result of ASIN, DASIN, QASIN, ACOS, DACOS, QACOS, ATAN, DATAN, QATAN, ATAN2, DATAN2, or QATAN2 is in radians. The result of ASIND, DASIND, QASIND, ACOSD, DACOSD, QACOSD, ATAND, DATAND, QATAND, ATAN2D, DATAN2D, or QATAN2D is in degrees.
6. If the value of the first argument of ATAN2, DATAN2, or QATAN2 is positive, the result is positive. When the value of the first argument is zero, the result is zero if the second argument is positive and π if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is $\pi/2$. Both arguments must not have the value zero. The range of the result for ATAN2, DATAN2, and QATAN2 is: $-\pi < \text{result} < \pi$.
7. If the value of the first argument of ATAN2D, DATAN2D, or QATAN2D is positive, the result is positive. When the value of the first argument is zero, the result will be zero if the second argument is positive and 180 degrees if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is 90 degrees. Both arguments must not have the value zero. The range of the result for ATAN2, DTAN2D, QATAN2D is: $-180 \text{ degrees} < \text{result} < 180 \text{ degrees}$.

Notes (Cont.)

8. The absolute value of a complex number, (X,Y), is the real value:
 $(X^2+Y^2)^{1/2}$
9. [x] is defined as the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as that of x. For example [5.7] equals 5. and [-5.7] equals -5.
10. Functions that cause conversion of one data type to another type provide the same effect as the implied conversion in assignment statements. The following functions return the value of the argument without conversion: the function REAL with a real argument, the function DBLE with a double precision argument, the function INT with an integer argument, and the function QEXT with a REAL*16 argument.
11. See Chapter 6 for additional information on character functions.
12. The functions INT, IDINT, IQINT, NINT, IDNINT, IQNINT, IFIX, MAX1, MINI, and ZEXT return INTEGER*4 values if the /I4 command qualifier is in effect, INTEGER*2 values if the /NOI4 qualifier is in effect.
13. When CMPLX and DCMPLX have only one argument, this argument is converted into the real part of a complex value, and zero is assigned to the imaginary part. (When there are two arguments (not complex), a complex value is produced by converting the first argument into the real part of the value and converting the second argument into the imaginary part.)
14. Bits in a_1 beyond the value specified by a_3 are unaffected.

D.4 System Subroutine Summary

The VAX FORTRAN system provides subroutines that you call in the same manner as a user-written subroutine. These subroutines are described in this section.

The subroutines supplied are:

DATE	Returns a 9-byte string containing the ASCII representation of the current date.
IDATE	Returns three integer values representing the current month, day, and year.
ERRSNS	Returns information about the most recently detected error condition.
EXIT	Terminates the execution of a program and returns control to the operating system.
SECNDS	Provides system time of day, or elapsed time, as a floating-point value in seconds.
TIME	Returns an 8-byte string containing the ASCII representation of the current time in hours, minutes, and seconds.

- RAN** Returns the next number from a sequence of pseudo random numbers of uniform distribution over the range 0 to 1.
- MVBITS** Transfers a bit field from one storage location to another.

References to integer arguments in the following subroutine descriptions refer to arguments of either INTEGER*4 data type or INTEGER*2 data type. However, the arguments must be either all INTEGER*4 or all INTEGER*2. In general, INTEGER*4 variables or array elements may be used as input values to these subroutines if their value is within the INTEGER*2 range.

D.4.1 DATE Subroutine

The DATE subroutine obtains the current date as set within the system. The call to DATE has the form:

```
CALL DATE(buf)
```

where:

buf

is a 9-byte variable, array, array element, or character substring. The date is returned as a 9-byte ASCII character string of the form:

```
dd-mmm-yy
```

where:

dd

is the 2-digit date.

mmm

is the 3-letter month specification.

yy

is the last two digits of the year.

D.4.2 IDATE Subroutine

The IDATE subroutine returns three integer values representing the current month, day, and year. The call to IDATE has the form:

```
CALL IDATE(i,j,k)
```

If the current date were October 9, 1984, the values of the integer variables upon return would be:

```
i = 10
```

```
j = 9
```

```
k = 84
```

D.4.3 ERRSNS Subroutine

The ERRSNS subroutine returns information about the most recent error that has occurred during program execution. The call to ERRSNS has the form:

```
CALL ERRSNS(fnum,rmssts,rmsstv,iunit,condval)
```

where:

fnum

is an integer variable or array element in which the most recent FORTRAN error number is stored. VAX FORTRAN error numbers are listed in Table 18-1.

A zero is returned if no error has occurred since the last call to ERRSNS, or if no error has occurred since the start of execution.

rmssts

if the last error was an RMS I/O error, is an integer variable or array element in which the RMS completion status code (STS) is stored.

rmsstv

if the last error was an RMS I/O error, is an integer variable or array element in which the RMS status value (STV) is stored. This status value provides additional status information.

iunit

if the last error was an I/O error, is an integer variable or array element in which the logical unit number is stored.

condval

is an integer variable or array element in which the actual VAX condition value is stored.

Any of the arguments can be null. If the arguments are of INTEGER*2 type, only the low-order 16 bits of information are returned. The saved error information is set to zero after each call to ERRSNS.

D.4.4 EXIT Subroutine

The EXIT subroutine causes program termination, closes all files, and returns control to the operating system. A call to EXIT has the form:

```
CALL EXIT[(exit-status)]
```

where:

exit-status

is an optional integer argument you can use to specify the image exit-status value.

D.4.5 SECNDS Subroutine

The SECNDS function subprogram returns the system time in seconds as a single-precision, floating-point value, minus the value of its single-precision, floating-point argument. The call to SECNDS has the form:

```
y = SECNDS(x)
```

where:

y

is set equal to the time in seconds since midnight, minus the user-supplied value of x.

The SECNDS function can be used to perform elapsed-time computations. For example:

```
C   START OF TIMED SEQUENCE
      T1 = SECNDS(0.0)

C   CODE TO BE TIMED

      DELTA = SECNDS(T1)
```

where DELTA will give the elapsed time.

The value of SECNDS is accurate to 0.01 second, which is the resolution of the system clock.

NOTE

1. The time is computed from midnight. SECNDS also produces correct results for time intervals that span midnight.
2. The 24 bits of precision provides accuracy to the resolution of the system clock for about one day. However, loss of significance can occur if you attempt to compute very small elapsed times late in the day. More precise timing information can be obtained using Run-Time Library procedures:

```
LIB$INIT__TIMER
LIB$SHOW__TIMER
LIB$STAT__TIMER
```

D.4.6 TIME Subroutine

The TIME subroutine returns the current system time as an ASCII string. The call to TIME has the form:

```
CALL TIME(buf)
```

where buf is an 8-byte variable, array, array element, or character substring.

The TIME call returns the time as an 8-byte ASCII character string of the form:

```
hh:mm:ss
```

where:

hh

is the 2-digit hour indication.

mm

is the 2-digit minute indication.

ss

is the 2-digit second indication.

For example:

10:45:23

A 24-hour clock is used.

D.4.7 RAN Subroutine

The RAN function is a general random number generator of the multiplicative congruential type. The result is a floating-point number that is uniformly distributed in the range between 0.0 inclusive and 1.0 exclusive. The call to RAN has the form:

$$y = \text{RAN}(i)$$

where:

y

is set equal to the value associated, by the function, with the argument *i*. The argument *i* must be an INTEGER*4 variable or INTEGER*4 array element.

The argument should initially be set to a large, odd integer value. The RAN function stores a value in the argument that it later uses to calculate the next random number.

There are no restrictions on the seed, although it should be initialized with different values on separate runs in order to obtain different random numbers. The seed is updated automatically, and RAN uses the following algorithm to update the seed passed as the parameter:

```
SEED = 69069 * SEED + 1 (MOD 2**32)
```

The value of SEED is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

D.4.8 MVBITS Subroutine

The MVBITS subroutine transfers a bit field from one storage location (source) to a field in a second storage location (destination). The call to MVBITS has the form:

```
CALL MVBITS(m,i,len,n,j)
```

where:

m

is an integer variable or array element that represents the source location, that is, the location from which a bit field is transferred.

i

is an integer expression that identifies the first bit position in the field transferred from m.

len

is an integer expression that identifies the length of the field transferred from m.

n

is an integer variable or array element that represents the destination location, that is, the location to which a bit field is transferred.

j

is an integer expression that identifies the bit in which the transferred bit field begins.

The MVBITS subroutine transfers len bits from positions i through i+len-1 of the source location (m) to positions j through j+len-1 of the destination location (n). Other bits of the destination location and all of the bits of the source location remain unchanged. The values of i+len must be less than 32, and j+len must be less than or equal to 32.

D.5 Bit Functions

VAX FORTRAN provides intrinsic functions for manipulation of the bits in the binary patterns that represent integer data types. For more information, refer to Table D-3.

D.5.1 Bit Position

Integer data types are represented internally in binary twos complement notation. Bit positions in the binary representation are numbered from right (least significant bit) to left (most significant bit); the rightmost bit position is numbered 0. A bit in a binary pattern has a value of 0 or 1.

D.5.2 Bit Function Arguments

The intrinsic functions IAND, IOR, IEOR, and NOT operate on all of the bits of their argument or arguments. Bit 0 of the result is the result of applying the specified logical operation to bit 0 of the argument or arguments. Bit 1 of the result is the result of applying the specified logical operation to bit 1 of the argument or arguments, and so on for all of the bits of the result.

The shift functions ISHFT and ISHFTC shift binary patterns. A positive shift count indicates a left shift, while a negative shift count indicates a right shift. ISHFT specifies a logical shift; bits shifted out of one end are lost and zeros are shifted in at the other end. ISHFTC performs a circular shift; bits shifted out at one end are shifted back in at the other end.

The functions IBSET, IBCLR, BTEST, and IBITS and the subroutine MVBITS operate on bit fields. A bit field is a contiguous group of bits within a binary pattern. Bit fields are specified by a starting bit position and a length. A bit field must be entirely contained in its source operand.

For example, the integer 47 is represented by the binary pattern:

0...0101111

bit position: n...6543210

where: n is the number of bit positions in the numeric storage unit

You can refer to the bit field contained in bits 3 through 6 by specifying a starting position of 3 and a length of 4.

Negative integers are represented in twos complement notation. The binary pattern for -47 is:

1...1010001

bit position: n...6543210

where: n is the number of bit positions in the numeric storage unit

In particular, note that the value of bit position n is 1 for a negative number and 0 for a non-negative number and that all of the high-order bits of the pattern from the last significant bit of the value up to bit n are the same as bit n.

IBITS and MVBITS operate on general bit fields. Both the starting position of a bit field and its length are arguments to these intrinsics. IBSET, IBCLR, and BTEST operate on 1-bit fields. They do not require a length argument.

For optimum selection of performance and memory requirements, FORTRAN provides two integer data types: INTEGER*2 requires two bytes of storage, while INTEGER*4 requires four bytes. The bit manipulation functions each have a generic form which operates on either of the two integer types and a specific form for each type. When you use the intrinsic functions which refer to bit positions or which shift binary patterns within a storage unit, you must be careful that you do not create a value that is outside the range of integers representable by the data type. For example:

```
INTEGER*2 I,J
I = 1
J = 17
I = ISHFT(I,J)
```

The variables I and J have INTEGER*2 data type. Therefore, the generic function ISHFT maps to the specific function IISHFT, which returns an INTEGER*2 result. INTEGER*2 results must be in the range -32768 to 32767, but the value 1, shifted left 17 positions, yields the binary pattern 1 followed by 17 zeros, which represents the integer 131072. Note that this would be valid if either I or J or both were INTEGER*4 because in both cases ISHFT would map to the specific function JISHFT, which returns an INTEGER*4 value.

If ISHFT is called with constant arguments, it returns an INTEGER*4 value.

Appendix E

Diagnostic Messages

Diagnostic messages related to a VAX FORTRAN program can come from the compiler, the linker, or the VAX run-time system. The compiler detects syntax errors in the source program, such as unmatched parentheses, invalid characters, misspelled keywords, and missing or invalid parameters. The run-time system reports errors that occur during execution.

This chapter lists and describes the messages issued by the compiler and the run-time system. It also provides a summary of the `DICTIONARY` messages that may accompany Common Data Dictionary messages. Linker messages are summarized in the *VAX/VMS Linker Reference Manual*.

E.1 Diagnostic Messages from the Compiler

A diagnostic message issued by the compiler describes the detected error, and in some cases contains an indication of the action taken by the compiler in response to the error.

Besides reporting errors detected in source program syntax, the compiler issues messages indicating errors that involve the compiler itself, such as I/O errors.

E.1.1 Source Program Diagnostic Messages

There are four classes of source program diagnostic messages. In order of greatest to least severity, these classes are:

Code	Description
F	Fatal; must be corrected before the program can be compiled. No object file is produced if an F-class error is detected during compilation.
E	Error; should be corrected. An object file is produced despite the E-class error, but the output or program result may be incorrect.

Code	Description
W	Warning; should be investigated by checking the statements to which W-class diagnostic messages apply. Warnings are issued for statements that use acceptable, but nonstandard, syntax and for statements corrected by the compiler. An object file is produced, but the program results may be incorrect. Note that W-class messages are produced unless the /NOWARNINGS qualifier is specified in the FORTRAN command.
I	Information; not an error message and does not call for corrective action. However, the I-class message informs you that either a correct VAX FORTRAN statement may have unexpected results or you have used a VAX extension to FORTRAN-77.

Typing mistakes are a likely cause of syntax errors: they can cause the compiler to generate misleading diagnostic messages. Beware especially of the following:

- Missing comma or parenthesis in a complicated expression or FORMAT statement.
- Misspelled variable names. The compiler may not detect this error, so execution can be affected.
- Inadvertent line continuation mark. This can cause a diagnostic message for the preceding line.
- Extension of the statement line past column 72. Unless /EXTEND_SOURCE is specified, this can cause diagnostic messages because the statement is terminated early.
- Confusion between the digit 0 and the uppercase letter O. This can result in variable names that appear identical to you but not to the compiler.

Another source of diagnostic messages is the inclusion of invalid ASCII characters in the source program. With the exception of the tab, space, and form-feed characters, nonprinting ASCII control characters are not valid in a FORTRAN source program. As the source program is scanned, such invalid characters are replaced by a question mark (?). However, because the question mark cannot occur in a FORTRAN statement, a syntax error usually results.

Because a diagnostic message indicates only the immediate cause, you should always check the entire source statement carefully.

The following examples show how source program diagnostic messages are displayed in interactive mode at your terminal. Figure E-1 shows how these messages appear in listings.

```

%FORT-W-FMTEXTCOM, Extra comma in format list
      [FORMAT (I3,)] in module MORTGAGE at line 13

%FORT-F-UNDSTALAB, Undefined statement label
      [66] in module MORTGAGE at line 19

%FORT-F-ENDNOOBJ, DB1:[SMITH]MOR.FOR;i completed
with 2 diagnostics - object deleted

0001   C          Program to calculate monthly mortgage payments
0002
0003          PROGRAM MORTGAGE
0004
0005          TYPE 10
0006   10        FORMAT ( ' ENTER AMOUNT OF MORTGAGE ' )
0007          ACCEPT 20, IPV
0008   20        FORMAT (I6)
0009
0010          TYPE 30
0011   30        FORMAT ( ' ENTER LENGTH OF MORTGAGE IN MONTHS ' )
0012          ACCEPT 40, IMON
0013   40        FORMAT (I3,)
0014
%FORT-W-FMTEXTCOM, Extra comma in format list
      [FORMAT (I3,)] in module MORTGAGE at line 13

0015          TYPE 50
0016   50        FORMAT ( ' ENTER ANNUAL INTEREST RATE ' )
0017          ACCEPT 60, YINT
0018   60        FORMAT (F6.4)
0019          GO TO 66
0020   65        YI = YINT/12    !Get monthly rate
0021          IMON = -IMON
0022          FIPV = IPV * YI
0023          YI = YI + 1
0024          FIMON = YI**IMON
0025          FIMON = 1 - FIMON
0026          FMNTHLY = FIPV/FIMON
0027
0028          TYPE 70, FMNTHLY
0029   70        FORMAT ( ' MONTHLY PAYMENT EQUALS ',F7.3 )
0030          STOP
0031          END
%FORT-F-UNDSTALAB, Undefined statement label
      [66] in module MORTGAGE at line 19

```

Figure E-1: Sample Diagnostic Messages (Listing Format)

Table E-1 is an alphabetical list of FORTRAN diagnostic error messages. For each message, the table gives a mnemonic, an error code level, the text of the message, and an explanation of the message.

Table E-1: Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
ADJARRBOU	E	Adjustable array bounds must be dummy arguments or in common Variables specified in dimension declarator expressions must either be subprogram dummy arguments or appear in common.
ADJARRUSE	F	Adjustable array used in invalid context A reference was made to an adjustable array in a context where such a reference is not allowed.
ADJLENUSE	F	Passed-length character name used in invalid context A reference was made to a passed-length character array or variable in a context where such reference is not allowed.
ALTRETLAB	F	Alternate return label used in invalid context An alternate return argument was used in a function reference.
ALTRETOMI	E	Alternate return omitted in SUBROUTINE or ENTRY statement An asterisk is missing in the argument list of a subroutine for which an alternate return is specified. Examples: <ol style="list-style-type: none"> 1. SUBROUTINE XYZ(A,B) <ul style="list-style-type: none"> · · · RETURN 1 2. ENTRY ABC(Q,R) <ul style="list-style-type: none"> · · · RETURN I

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
ALTRETSPE	F	<p>Alternate return specifier invalid in FUNCTION subprogram</p> <p>The argument list of a FUNCTION declaration contains an asterisk, or a RETURN statement in a function subprogram specifies an alternate return. Examples:</p> <ol style="list-style-type: none"> 1. INTEGER FUNCTION TCB(ARG,*,X) 2. FUNCTION IMAX . . . RETURN I+J END
ARIVALREQ	F	<p>Character expression where arithmetic value required</p> <p>An expression that must be arithmetic (INTEGER, REAL, LOGICAL, or COMPLEX) was of type CHARACTER.</p>
ASSARRUSE	F	<p>Assumed size array name used in invalid context</p> <p>An assumed size array name was used where the size of the array was also required, for example, in an I/O list.</p>
ASSDOVAR	W	<p>Assignment to DO variable within loop</p> <p>The control variable of a DO loop has been altered within the range of the DO statement.</p>
BADEND	F	<p>END [STRUCTURE UNION MAP] must match top.</p> <p>A STRUCTURE, UNION, or MAP statement did not have a corresponding END STRUCTURE, END UNION, or END MAP statement, respectively.</p>
BADFIELD	F	<p>Field name not defined for this structure.</p> <p>A field name not defined in a structure was used in a qualified reference.</p>
BADRECFEF	F	<p>Aggregate reference where scalar reference required</p> <p>An aggregate reference was used where a scalar reference was required.</p>
CDDBITSIZ	F	<p>CDD field specifies a bit size or alignment. Size or address rounded up to byte alignment.</p> <p>CDD's bit datatype and bit alignment are not supported by FORTRAN.</p>

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
CDDERROR	I	CDD description extraction condition The FORTRAN compiler is in the process of extracting a data definition from the Common Data Dictionary. See the accompanying messages for more information.
CDDNOTSTR	F	CDD record is not a structure CDD record description was not structured. VAX FORTRAN requires structure definitions (elementary field descriptions in CDDL).
CDDRECDIM	F	CDD record is dimensioned VAX FORTRAN does not support dimensioned structures, for example, arrays of structures.
CDDSCALED	W	CDD description specifies a scaled data type VAX FORTRAN does not support scaled data types. The data described by the CDD specifies a scaled component.
CDDTOOBIG	E	Attributes for some member of CDD record description exceed implementation's limit for member complexity Some member of the CDD record description has too many attributes and has created a program that is too large. Change the Common Data Dictionary description to make the field description smaller.
CDDTOODEEP	E	Attributes for CDD record description exceed implementation's limit for record complexity The CDD record description contains structures that are nested too deeply. Modify the CDD description to reduce the level of nesting in the record description.
CHANAMINC	E	Character name incorrectly initialized with numeric value Character data with a length greater than one was initialized with a numeric value in a data statement. Example: CHARACTER*4 A DATA A/14/
CHASBSLIM	F	Character substring limits out of order The first character position of a substring expression is greater than the last character position. Example: C(5:3)

Table E-1.(Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
CHAVALREQ	F	Arithmetic expression where character value required An expression that must be of type CHARACTER was of another data type.
COLMAJOR	F	CDD description specifies that it is not a column major array FORTRAN only supports column-major arrays. Change the CDD description to specify a column-major array.
CONSIZEXC	E	Constant size exceeds variable size in data initialization A constant used for data initialization is larger than its corresponding variable.
DBGOPT	I	The NOOPTIMIZE qualifier is recommended with the DEBUG qualifier. Optimizations performed by the compiler can cause several different kinds of unexpected behavior when using VAX DEBUG. See Chapter 1 of <i>VAX FORTRAN User's Guide</i> for more information.
DEFSTAUNK	I	Default STATUS='UNKNOWN' used in OPEN statement The OPEN statement default STATUS='UNKNOWN' may cause an old file to be inadvertently modified.
DEPENDITEM	I	CDD description contains Depends Item attribute (ignored). FORTRAN does not support the CDD Depends Item attribute. No action is required.
DICTABORT	F	DICTIONARY processing of CDD record description aborted The FORTRAN compiler is unable to process the CDD record description. See the accompanying messages for further information.
ENTDUMVAR	F	ENTRY dummy variable previously used in executable statement The dummy arguments of an ENTRY statement must not have been used previously in an executable statement in the same program unit.
EQVEXPCOM	F	EQUIVALENCE statement incorrectly expands a common block A common block cannot be extended beyond its beginning by an EQUIVALENCE statement.
EXCCHATRU	E	Non-blank characters truncated in string constant A character or Hollerith constant was converted to a data type which was not large enough to contain all the significant characters.

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
EXCDIGTRU	E	Non-zero digits truncated in hex or octal constant An octal or hexadecimal constant was converted to a data type which was not large enough to contain all the significant digits.
EXCNAMDAT	E	Number of names exceeds number of values in data initialization The number of constants specified in a DATA statement must match the number of variables or array elements to be initialized. The remaining variables and array elements are not initialized.
EXCVALDAT	E	Number of values exceeds number of names in data initialization The number of variables or array elements to be initialized must match the number of constants specified in data initialization. The remaining constant values are ignored.
EXPSTAOVE	F	Compiler expression stack overflow An expression is too complex or there are too many actual arguments in a subprogram reference. A maximum of 255 actual arguments can be compiled. You can subdivide a complex expression or reduce the number of arguments.
EXTCHAFOL	E	Extra characters following a valid statement Superfluous text was found at the end of a syntactically correct statement. Check for typing or syntax errors.
EXTMIXCOM	I	Extension to FORTRAN-77: Mixed numeric and character elements in common Numeric and character variable and array elements cannot be equivalenced to each other.
EXTMIXEQV	I	Extension to FORTRAN-77: Mixed numeric and character elements in EQUIVALENCE A common block must not contain both numeric and character data.
EXTRECUSE	I	Extension to FORTRAN-77: Nonstandard use of field reference. A record reference (for example, record-name.field-name) was used in a program compiled with the /STANDARD=[SYNTAX_ALL] qualifier in the FORTRAN command.
EXT_COM	I	Extension to FORTRAN-77: nonstandard comment FORTRAN-77 allows only the characters "C" and "*" to begin a comment line; "D", "d", and "!" are extensions to FORTRAN-77.

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
EXT_CONST	I	<p>Extension to FORTRAN-77: nonstandard constant</p> <p>The following constant forms are extensions to FORTRAN-77:</p> <ul style="list-style-type: none"> • Hollerith nH..... • Typeless 'xxxx'X or 'oooo'O • Octal "oooo or Ooooo • Hexadecimal Zxxxx • Radix-50 nR..... • Complex with PARAMETER components • COMPLEX*16 (www.xxxDn, yyy.zzzDn) • REAL*16 yyy.zzzQn
EXT_FMT	I	<p>Extension to FORTRAN-77: nonstandard FORMAT statement item</p> <p>The following format field descriptors are extensions to FORTRAN-77:</p> <ul style="list-style-type: none"> • \$,O,Z All forms • A,L,I,F,E,G,D Default field width forms • P Without scale factor
EXT_KEY	I	<p>Extension to FORTRAN-77: nonstandard keyword</p> <p>A nonstandard keyword was used.</p>
EXT_LEX	I	<p>Extension to FORTRAN-77: nonstandard lexical item</p> <p>One of the following nonstandard lexical items was used:</p> <ul style="list-style-type: none"> • An alternate return specifier with an ampersand (&) in a CALL statement • The apostrophe (') form of record specifier in a direct access I/O statement • A variable format expression
EXT_NAME	I	<p>Extension to FORTRAN-77: nonstandard name</p> <p>A name longer than six characters or one that contained a dollar sign (\$) or an underscore (_) was used.</p>

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
EXT_OPER	I	<p>Extension to FORTRAN-77: nonstandard operator</p> <p>The operators .XOR., %VAL, %REF, %DESCR, and %LOC are extensions to FORTRAN-77. The standard form of .XOR. is .NEQV. The % operators are extensions provided to allow access to non-FORTRAN parts of the VAX-11 environment.</p>
EXT_SOURC	I	<p>Extension to FORTRAN-77: tab indentation or lowercase source</p> <p>The use of tab indentation or lowercase letters in source code is an extension to FORTRAN-77.</p>
EXT_STMT	I	<p>Extension to FORTRAN-77: nonstandard statement type</p> <p>A nonstandard statement type was used.</p>
EXT_SYN	I	<p>Extension to FORTRAN-77: nonstandard syntax</p> <p>One of the following syntax extensions was specified:</p> <ul style="list-style-type: none"> • PARAMETER name = value No parentheses • type name/value/ Data initialization in type declaration • DATA (ch(exp:exp),v=e2)/values/ Substring initialization with implied-DO in DATA statement • CALL name(arg2,,arg3) Null actual argument • READ (...),iolist Comma between I/O control and element lists • PARAMETER (name2=ABS(name1)) Function use in PARAMETER • e1 ** -e2 Two consecutive operators

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning																										
EXT_TYPE	I	<p>Extension to FORTRAN-77; nonstandard data type specification</p> <p>The following DATA type specifications are extensions to FORTRAN-77. The FORTRAN-77 equivalent is given where available. This message is issued when these types are used in the IMPLICIT statement or in a numeric type statement.</p> <table border="0"> <thead> <tr> <th align="left">Extension</th> <th align="left">Standard</th> </tr> </thead> <tbody> <tr> <td>BYTE</td> <td></td> </tr> <tr> <td>LOGICAL*1</td> <td></td> </tr> <tr> <td>LOGICAL*2</td> <td>LOGICAL (with /NOI4 specified only)</td> </tr> <tr> <td>LOGICAL*4</td> <td>LOGICAL</td> </tr> <tr> <td>INTEGER*2</td> <td>INTEGER (with /NOI4 specified only)</td> </tr> <tr> <td>INTEGER*4</td> <td>INTEGER</td> </tr> <tr> <td>REAL*4</td> <td>REAL</td> </tr> <tr> <td>REAL*8</td> <td>DOUBLE PRECISION</td> </tr> <tr> <td>REAL*16</td> <td></td> </tr> <tr> <td>COMPLEX*8</td> <td>COMPLEX</td> </tr> <tr> <td>COMPLEX*16</td> <td></td> </tr> <tr> <td>DOUBLE COMPLEX</td> <td></td> </tr> </tbody> </table>	Extension	Standard	BYTE		LOGICAL*1		LOGICAL*2	LOGICAL (with /NOI4 specified only)	LOGICAL*4	LOGICAL	INTEGER*2	INTEGER (with /NOI4 specified only)	INTEGER*4	INTEGER	REAL*4	REAL	REAL*8	DOUBLE PRECISION	REAL*16		COMPLEX*8	COMPLEX	COMPLEX*16		DOUBLE COMPLEX	
Extension	Standard																											
BYTE																												
LOGICAL*1																												
LOGICAL*2	LOGICAL (with /NOI4 specified only)																											
LOGICAL*4	LOGICAL																											
INTEGER*2	INTEGER (with /NOI4 specified only)																											
INTEGER*4	INTEGER																											
REAL*4	REAL																											
REAL*8	DOUBLE PRECISION																											
REAL*16																												
COMPLEX*8	COMPLEX																											
COMPLEX*16																												
DOUBLE COMPLEX																												
FLDNAME	F	<p>Structure field is missing a field name.</p> <p>Unnamed fields are not allowed. The effect of an unnamed field can be achieved by using the pseudo-name %FILL in place of a field name in a typed data declaration.</p>																										
FMTEXTCOM	W	<p>Extra comma in format list</p> <p>Example: FORMAT (I4,)</p>																										
FMTEXTNUM	E	<p>Extra number in format list</p> <p>Example: FORMAT (I4,3)</p>																										
FMTINVCHA	E	<p>Format item contains meaningless character</p> <p>An invalid character or a syntax error was detected in a FORMAT statement.</p>																										
FMTINVCON	E	<p>Constant in format item out of range</p> <p>A numeric value in a FORMAT statement exceeds the allowable range. Refer to Chapter 12 for information about range limits.</p>																										
FMTMISNUM	E	<p>Missing number in format list</p> <p>Example: FORMAT (F6.)</p>																										

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
FMTMISSEP	E	Missing separator between format items A required separator character has been omitted between fields in a FORMAT statement.
FMTNEST	E	Format groups nested too deeply Format groups cannot be nested beyond eight levels.
FMTPAREN	E	Unbalanced parentheses in format list The number of right parentheses does not match the number of left parentheses.
FMTSIGN	E	Format item cannot be signed A signed constant is valid only with the P format code.
HOLCOURED	E	Count of Hollerith or Radix-50 constant too large, reduced The value specified by the integer preceding the H or R is greater than the number of characters remaining in the source statement.
IDOINVOP	F	Invalid operation in implied-DO list An invalid operation was attempted in an implied-DO list in a DATA statement, for example, a function reference in the subscript or substring expression of an array or character substring reference. Example: DATA (A(SIN(REAL(I))), I=1,10) /101./
IDOINVPAR	F	Invalid DO parameters in implied-DO list An invalid control parameter was detected in an implied-DO list in a DATA statement, for example, an increment of zero.
IDOINVREF	F	Invalid reference to name in implied-DO list A control parameter expression in an implied-DO list in a DATA statement contains a name which is not the name of a control variable of any implied-DO list which has the name in its scope. Example: DATA (A(J), J=1,10), (B(I), I=J,K) /1001./ Both J and K in the second implied-DO list are invalid names.

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
IDOSYNERR	F	Syntax error in implied-DO list in data initialization Improper syntax was detected in an implied-DO list in data initialization, for instance, improperly nested parentheses.
IMPDECLAR	W	Use of implicit with declaration warnings. An IMPLICIT statement was used in a program compiled with the /WARNINGS=DECLARATIONS qualifier in the FORTRAN command.
IMPMULTYP	E	Letter mentioned twice in IMPLICIT statement, last type used A letter has been given an implicit data type more than once. The last data type given is used.
IMPNONE	E	Untyped name, must be explicitly typed The displayed name has not been defined in any data type declaration statement, and IMPLICIT NONE statement has been specified. Check that the name was not accidentally created by an undetected syntax error. Example: DO 10 I = 1,10 The apparent DO statement is really an assignment to the accidentally created variable DO10I.
IMPSYNERR	E	Syntax error in IMPLICIT statement Improper syntax was used in an IMPLICIT statement. Refer to Section 8.8 for the syntax rules.
INCDONEST	F	DO or IF statement incorrectly nested One of the following conditions was found: <ul style="list-style-type: none"> • A statement label specified in a DO statement has been used previously. Example: <pre> 10 I = I + 1 J = J + 1 DO 10 K=1,10 . . . </pre>

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
INCDONEST (Cont.)	F	<ul style="list-style-type: none"> • A DO loop contains an incomplete DO loop or IF block. Examples: <pre> 1. DO 10 I=1,10 J = J + 1 DO 20 K=1,10 J = J + K 10 CONTINUE The start of the incomplete IF block can be a block IF, ELSE IF, or ELSE statement. 2. DO 10 I=1,10 J = J + I IF (J .GT. 20) THEN J = J - 1 ELSE J = J + 1 10 CONTINUE END IF </pre>
INCFILNES	F	<p>INCLUDE files and/or DICTIONARY statements nested too deeply</p> <p>Up to 10 levels of nested INCLUDE files and/or DICTIONARY statements are permitted.</p>
INCFUNTYP	F	<p>Inconsistent function data types</p> <p>The function name and entry points in a function subprogram must be consistent within one of three groups of data types:</p> <p>Group 1: All numeric types except REAL*16, COMPLEX*16</p> <p>Group 2: REAL*16, COMPLEX*16</p> <p>Group 3: Character</p> <p>Example:</p> <pre> CHARACTER*15 FUNCTION I REAL*4 G ENTRY G </pre>
INCLABUSE	F	<p>Inconsistent usage of statement label</p> <p>Labels of executable statements have been confused with labels of FORMAT statements or with labels of nonexecutable statements.</p> <p>Example:</p> <pre> GD TO 10 10 FORMAT (I5) </pre>

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
INCLENMOD	F	<p>Incorrect length modifier in declaration</p> <p>An unacceptable length has been specified in a data type declaration (see Section 8.4). For example:</p> <pre>INTEGER PIPES*8</pre>
INCMODNAM	F	<p>Module name not found in library</p> <p>The module name specified in an INCLUDE statement could not be located in the specified library. Check the name of the module and library.</p>
INCOPEFAI	F	<p>Open failure on INCLUDE file</p> <p>The specified file could not be opened, possibly due to an incorrect file specification, nonexistent file, unmounted volume, or a protection violation.</p>
INCSTAFUN	E	<p>Inconsistent statement function reference</p> <p>The actual argument(s) in a statement function reference do not agree in either order, number, or data type with the formal arguments declared.</p>
INCSYNERR	F	<p>Syntax error in INCLUDE file specification</p> <p>The file-name string is not acceptable (invalid syntax, invalid qualifier, undefined device, and so on).</p>
INQUNIT	F	<p>Missing or invalid use of UNIT or FILE specifier in INQUIRE statement.</p> <p>An INQUIRE statement must have a UNIT specifier or a FILE specifier, but may not have both.</p>
INTFUNARG	E	<p>Arguments incompatible with intrinsic function, assumed EXTERNAL</p> <p>A function reference was made, using an intrinsic function name, but the argument list does not agree in order, number, or type with the intrinsic function requirements. The function is assumed to be supplied by you as an EXTERNAL function.</p>
INTVALREQ	F	<p>Non-integer expression where integer value required</p> <p>An expression that must be of type INTEGER was another data type.</p>
INVACTARG	E	<p>Invalid use of intrinsic function name as actual argument</p> <p>A generic intrinsic function name was used as an actual argument.</p>

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
INVASSVAR	E	Invalid ASSOCIATEVARIABLE specification An ASSOCIATEVARIABLE specification in an OPEN or DEFINE FILE statement was a dummy argument or an array element.
INVCHAUSE	E	Invalid character used in constant An invalid character was detected in a constant. Valid characters are: Hexadecimal: 0 - 9, A - F, a - f Octal: 0 - 7 Radix-50: A - Z, 0 - 9, \$, period, or space For Radix-50, a space is substituted for the invalid character. For hexadecimal and octal, the entire constant is set to zero.
INVCONST	E	Arithmetic error while evaluating constant or constant expression The specified value of a constant is too large or too small to be represented.
INVCONSTR	F	Invalid control structure using ELSE IF, ELSE, or END IF The order of ELSE IF, ELSE, or END IF statements is incorrect. ELSE IF, ELSE, and END IF statements cannot stand alone. ELSE IF and ELSE must be preceded by either a block IF statement or an ELSE IF statement. END IF must be preceded by either a block IF, ELSE IF, or ELSE statement. Examples: 1. DO 10 I=1,10 J = J + I ELSE IF (J ,LE. K) THEN Error: ELSE IF preceded by a DO statement. 2. IF (J ,LT. K) THEN J = I + J ELSE J = I - J ELSE IF (J ,EQ. K) THEN END IF Error: ELSE IF preceded by an ELSE statement.
INVDOTERM	W	Statement cannot terminate a DO loop The terminal statement of a DO loop cannot be a GO TO, arithmetic IF, RETURN, block IF, ELSE, ELSE IF, END IF, DO, or END statement.

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
INVENDKEY	W	Invalid END= keyword, ignored The END keyword was used illegally in a WRITE, REWRITE, direct access READ, or keyed access READ statement.
INVENTORY	E	ENTRY within DO loop or IF block, statement ignored An ENTRY statement is not allowed within the range of a DO loop or IF block.
INVEQVCOM	F	Invalid equivalence of two variables in common Variables in common cannot be equivalenced to each other.
INVFUNUSE	F	Invalid use of function name in CALL statement A CALL statement referred to a subprogram name that was used as a CHARACTER, REAL*16, or COMPLEX*16 function. Example: IMPLICIT CHARACTER*10(C) CSCAL = CFUNC(X) CALL CFUNC(X)
INVINIVAR	E	Invalid initialization of variable not in common An attempt was made, in a BLOCK DATA subprogram, to initialize a variable that is not in a common block.
INVINTFUN	E	Name used in INTRINSIC statement is not an intrinsic function A function name which appeared in the INTRINSIC statement is not an intrinsic function.
INVIOSPEC	F	Invalid I/O specification for this type of I/O statement A syntax error was found in the portion of an I/O statement that precedes the I/O list. Examples: 1. TYPE (G), J 2. WRITE 100, J
INVKEYOPE	F	Incorrect keyword in OPEN, CLOSE, or INQUIRE statement An OPEN, CLOSE, or INQUIRE statement contains a keyword which is not valid for that statement.

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
INVLEFSID	F	Left side of assignment must be variable or array element The symbolic name to which the value of an expression is assigned must be a variable, array element, or character substring reference.
INVLEXEME	F	Variable name, constant, or expression invalid in this context An entity has been used incorrectly; for example, the name of a subprogram was used where an arithmetic expression is required.
INVLOGIF	F	Statement cannot appear in logical IF statement A logical IF statement must not contain a DO statement or another logical IF, IF THEN, ELSE IF, ELSE, END IF, or END statement.
INVNMLELE	F	Invalid NAMELIST element Dummy argument or element other than variable or array name appeared in NAMELIST declaration.
INVNUMSUB	F	Number of subscripts does not match array declaration More or fewer dimensions than were declared for the array are referenced.
INVPERARG	F	Invalid argument to %VAL, %REF, %DESCR, or %LOC The argument specified for one of the built-in functions is not valid. Examples: 1. %VAL (3.5D0) — Argument cannot be REAL*8, REAL*16, character, or complex. 2. %LOC (X+Y) — Argument must not be an expression.
INVPERUSE	E	%VAL, %REF, or %DESCR used in invalid context The argument list built-in functions (%VAL, %REF, %DESCR) cannot be used outside an actual argument list. Example: X = %REF(Y)
INVQUAL	I	Invalid qualifier or qualifier value in OPTIONS statement An invalid qualifier or qualifier value was specified in the OPTIONS statement. The qualifier is ignored.

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
INVRECUSE	F	<p>Invalid use of record or array name</p> <p>A statement in the program violated one of the following rules:</p> <ul style="list-style-type: none"> • An aggregate cannot be assigned to a nonaggregate or to an aggregate with a structure that isn't the same. • An array name reference cannot be qualified. • Aggregate references cannot be used in I/O lists of formatted I/O statements.
INVREPCOU	E	<p>Invalid repeat count in data initialization, count ignored</p> <p>The repeat count in a data initialization was not an unsigned, nonzero integer constant. The count is ignored.</p>
INVSBSREF	E	<p>Substring reference used in invalid context</p> <p>A substring reference to a variable or array that is not of type CHARACTER has been detected. Example:</p> <pre>REAL X(10) Y = X(J:K)</pre>
INVSTALAB	W	<p>Invalid statement label ignored</p> <p>An improperly formed statement label (namely, a label containing letters) has been detected in columns 1 to 5 of an initial line. The statement label is ignored.</p>
INVSUBREF	F	<p>Subscripted reference to non-array variable</p> <p>A variable that is not defined as an array cannot appear with subscripts.</p>
INVTYPUSE	F	<p>Name previously used with conflicting data type</p> <p>A data type was assigned to a name that had already been used in a context that required a different data type.</p>
IODUPKEY	F	<p>Duplicated keyword in I/O statement</p> <p>Each keyword subparameter in an I/O statement or auxiliary I/O statement can be specified only once.</p>

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
IOINVFMT	F	<p>Format specifier in error</p> <p>The format specifier in an I/O statement is invalid. It must be one of the following:</p> <ul style="list-style-type: none"> • The label of a FORMAT statement. • An asterisk (*). (List-directed I/O.) • A run-time format specifier: variable, array element, or character substring reference. • An integer variable that has been assigned a FORMAT label by an ASSIGN statement.
IOINVKEY	F	<p>Invalid keyword for this type of I/O statement</p> <p>An I/O statement contains a keyword which cannot be used with that type of I/O statement.</p>
IOINVLIST	F	<p>Invalid I/O list element for input statement</p> <p>An input statement I/O list contains an invalid element, such as an expression or a constant.</p>
IOSYNERR	F	<p>Syntax error in I/O list</p> <p>Improper syntax was detected in an I/O list.</p>
LABASSIGN	F	<p>Label in ASSIGN statement exceeds INTEGER*2 range</p> <p>A label whose value is assigned to an INTEGER*2 variable by an ASSIGN statement must not be separated by more than 32K bytes from the beginning of the code for the program unit.</p>
LENCHAFUN	E	<p>Length specified must match CHARACTER FUNCTION declaration</p> <p>The length specifications for all ENTRY names in a character function subprogram must be the same. Example:</p> <pre>CHARACTER*15 FUNCTION F CHARACTER*20 G ENTRY G</pre>
LOGVALREQ	F	<p>Non-logical expression where logical value required</p> <p>An expression that must be of type LOGICAL was of another data type.</p>

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
LOWBOUGRE	E	Lower bound greater than upper bound in array declaration The upper bound of a dimension declarator must be equal to or greater than the lower bound.
MINDIGITS	W	CDD description specifies precision less than allowed for data type. Minimum precision has been supplied. Some Common Data Dictionary data types specify a number of digits which is incompatible with FORTRAN data types. The FORTRAN compiler has expanded the data type to conform to a FORTRAN data type. No action required.
MINOCCURS	I	CDD description contains Minimum Occurs attribute (ignored). FORTRAN does not support the Common Data Dictionary Minimum Occurs attribute. No action required.
MISSAPOS	E	Missing apostrophe in character constant A character constant must be enclosed by apostrophes.
MISSCONST	F	Missing constant A required constant was not found.
MISSDEL	F	Missing operator or delimiter symbol Two terms of an expression are not separated by an operator, or a punctuation mark (such as a comma) has been omitted. Examples: 1. CIRCUM = 3.14 DIAM 2. IF (I 10,20,30
MISSEND	E	Missing END statement, END is assumed An END statement was missing at the end of the last input file, and it has been inserted.
MISSEXPO	E	Missing exponent after E, D, or Q A floating-point constant was specified in E, D, or Q notation, but the exponent was omitted.
MISSKEY	F	Missing keyword A required keyword, such as TO, was omitted from a statement such as ASSIGN 10 TO I.
MISSLABEL	F	Missing statement label A required statement label reference was omitted.

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
MISSNAME	F	Missing variable or subprogram name A required variable name or subprogram name was not found.
MISSUNIT	F	Unit specifier keyword missing in I/O statement An I/O statement must include a unit specifier subparameter.
MISSVAR	F	Missing variable or constant An expression, or a term of an expression, has been omitted. Examples: 1. WRITE () 2. DIST = *TIME
MULDECNAM	F	Multiple declaration of name A name appears in two or more inconsistent declaration statements.
MULDECTYP	E	Multiple declaration of data type for variable, first type used A variable appears in more than one data type declaration statement. The first type declaration is used.
MULDEFLAB	E	Multiple definition of statement label, second ignored The same label appears on more than one statement. The first occurrence of the label is used.
MULFLDNAM	F	Multiply defined field name Each field name within the same level of a given structure declaration must be unique.
MULSTRNAM	F	Multiply defined STRUCTURE name A STRUCTURE name must be unique among STRUCTURE names.
NAMTOOLON	W	Name longer than 31 characters A symbolic name has been truncated to 31 characters.
NMLIOLIST	E	I/O list not permitted with namelist I/O An I/O statement with a namelist specifier incorrectly contained an I/O list.

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
NODFLOAT	W	<p>CDD description specifies the D__Floating data type. The data cannot be represented when compiling /G__FLOAT.</p> <p>D__floating datatype was specified when compiling with /G__FLOATING qualifier. Ignore the warning message or recompile the program using the /NOG__FLOATING qualifier.</p>
NOGFLOAT	W	<p>CDD description specifies G__Floating data type. The data cannot be represented when compiling /NOG__FLOAT.</p> <p>G__floating datatype was specified when compiling with /NOG__FLOATING qualifier. Ignore the warning message or recompile the program using the /G__FLOAT qualifier.</p>
NOHFLOAT	W	<p>CDD description specifies H__Floating data type. The data cannot be represented when compiling /NOG__FLOAT.</p> <p>H__floating datatype was specified when compiling with /NOG__FLOATING qualifier. Ignore the warning message or recompile the program using the /G__FLOATING qualifier.</p>
NONCONSUB	F	<p>Non-constant subscript where constant required</p> <p>Subscript and substring expressions used in DATA and EQUIVALENCE statements must be constants.</p>
NOPATH	W	<p>No path to this statement</p> <p>Program control cannot reach this statement. The statement is deleted. Example:</p> <pre> 10 I = I + 1 GO TO 10 STOP </pre>
OPEDOLOOP	F	<p>Unclosed DO loop or IF block</p> <p>The terminal statement of a DO loop or the END IF statement of an IF block was not found. Example:</p> <pre> DO 20 I=1,10 X = Y END </pre>
OPENOTPER	F	<p>Operation not permissible on these data types</p> <p>An invalid operation was specified, such as an .AND. of two real variables.</p>

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
PROSTOREQ	F	Program storage requirements exceed addressable memory The storage space allocated to the variables and arrays of the program unit exceeds the addressing range of the machine.
REDCONMAR	W	Redundant continuation mark ignored A continuation mark was detected where an initial line is required. The continuation mark is ignored.
REFERENCE	I	CDD description contains Reference attribute (ignored). The Reference attribute is not supported by FORTRAN. No action required.
SOURCETYPE	I	CDD description contains Source Type attribute (ignored). FORTRAN does not support the Common Data Dictionary Source Type attribute. No action required.
STAENDSTR	F	Statement not allowed within structure; structure definition closed A statement not allowed in a structure declaration block was encountered. The compiler assumes that you omitted one or more END STRUCTURE statements.
STAINVSTR	E	Statement not allowed within structure definition; statement ignored A statement not allowed in a structure declaration block was encountered. Structure declaration blocks can only include the following statements: typed data declaration statements, RECORD statements, UNION/END UNION statements, MAP/END MAP statements, and STRUCTURE/END STRUCTURE statements.
STANOTVAL	E	Statement not valid in this program unit, statement ignored A program unit contains a statement that is not allowed; for example, a BLOCK DATA subprogram contains an executable statement.
STAOUTORD	E	Statement out of order, statement ignored The order of statements was not as specified in Section 5.2.2.1. The statement found to be out of order is ignored.
STATOOCOM	F	Statement too complex A statement is too complex to be compiled. It must be subdivided into two or more statements.

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
STRCONTRU	E	String constant truncated to maximum length A character or Hollerith constant can contain up to 2000 characters. A Radix-50 constant can contain up to 12 characters.
STRDEPTH	F	STRUCTUREs/UNIONs/MAPs nested too deeply The combined nesting level limit for structures, unions, and maps is 20 levels.
STRNAME	E	Outer level structure is missing a structure name An outer level STRUCTURE statement must have a structure name in order for a RECORD statement to be able to reference the structure declaration.
STRNOTDEF	F	Structure name in RECORD statement not defined Either a RECORD statement did not contain a structure name enclosed within slashes or the structure name contained in a RECORD statement was not defined in a structure declaration.
SUBEXPVAL	E	Subscript or substring expression value out of bounds An array element beyond the specified dimensions, or a character substring outside the specified bounds, was referenced.
TAGVARIAB	I	CDD description contains Tag Variable attribute (ignored). FORTRAN does not support the Common Data Dictionary Tag Variable attribute. No action required.
TOOMANCOM	F	Too many named common blocks Reduce the number of named common blocks.
TOOMANCON	E	Too many continuation lines, remainder ignored Up to 99 continuation lines are permitted, as determined by the /CONTINUATIONS=n qualifier (default, 19).
TOOMANDIM	E	More than 7 dimensions specified, remainder ignored An array can be defined as having up to seven dimensions.
TOOMANYDO	F	DO and IF statements nested too deeply DO loops and block IF statements cannot be nested beyond 20 levels.

Table E-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Meaning
UNDARR	F	Undimensioned array or statement function definition out of order Either a statement function definition was found among executable statements or an assignment statement involving an undimensioned array was found.
UNDSTALAB	F	Undefined statement label Reference has been made to a statement label that is not defined in the program unit.
UNSUPPTYPE	I	CDD description specifies an unsupported data type. The Common Data Dictionary description for a structure item has attempted to use a data type that is not supported by FORTRAN. The FORTRAN compiler makes the data type accessible by declaring it as an inner structure containing a single CHARACTER %FILL field with an appropriate length. Change the data type to one that is supported by FORTRAN or use the FORTRAN built-in functions to manipulate the contents of the field.
VARINCEQV	F	Variable inconsistently equivalenced to itself EQUIVALENCE statements specify inconsistent relationships between variables or array elements. Example: EQUIVALENCE (A(1), A(2))
ZERLENSTR	E	Zero-length string The length specified for a character, Hollerith, hexadecimal, octal, or Radix-50 constant must not be zero.

E.1.2 Compiler-Fatal Diagnostic Messages

Conditions can be encountered of such severity that compilation must be terminated at once. These conditions are caused by hardware errors, software errors, and errors that require changing the FORTRAN command. Printed messages have the form:

FORT-F-error name, error message

The first line of the message contains the appropriate file specification or keyword involved in the error. The operating system supplies more specific information about the error whenever possible. For example, a file read error might produce the following error message:

```
%FORT-F-READERR, error reading _DBA0:[SMITH]MAIN.FOR;3  
-RMS-W-RTB, 512 byte record too big for user's buffer  
-FORT-F-ABORT, abort
```

Table E-2 lists the diagnostic messages that report the occurrence of such compiler-fatal errors. Because the exact content of the message depends upon the individual problem, only the first line of the message is provided here. Also, "file-spec" represents placement of the actual file specification in the message, and "keyword-value" represents the specific keyword value.

Table E-2: Compiler-Fatal Diagnostic Messages

I/O Errors

FORT-F-OPENIN, error opening "file-spec" as input

FORT-F-NOSOUFFILE, no source file specified

FORT-F-OPENOUT, error opening "file-spec" as output

FORT-F-READERR, error reading "file-spec"

FORT-F-WRITEERR, error writing "file-spec"

FORT-F-CLOSEIN, error closing "file-spec" as input

FORT-F-CLOSEOUT, error closing "file-spec" as output

Command Qualifier Messages

FORT-F-VALERR, specified value is out of legal range

FORT-F-BADVALUE, "keyword-value" is an invalid keyword value

FORT-F-SUBNOTALL, subqualifier not allowed with negated qualifier

Compiler Internal Logic Error

FORT-F-BUGCHECK, internal consistency failure

If you receive the compiler internal logic error, FORT-F-BUGCHECK, you should report both the error and the circumstance in which it occurred to DIGITAL by means of a Software Performance Report (SPR).

E.1.3 Compiler Limits

There are limits to the size and complexity of a single VAX FORTRAN program unit. There are also limits on the complexity of FORTRAN statements. Table E-3 describes some of these limits.

Table E-3: Compiler Limits

Language Element	Limit
Structure nesting	20
DO and block IF statement nesting (combined)	20
Actual number of arguments per CALL or function reference	255
Named common blocks	250
Format group nesting	8
Labels in computed or assigned GO TO list	500
Parentheses nesting in expressions	40
INCLUDE file nesting	10
Continuation lines	99
FORTTRAN source line length	132 characters
Symbolic name length	31 characters
Constants	
Character, Hollerith	2000 characters
Radix-50	12 characters
Array dimensions	7
Number of names in a NAMELIST group	250

The amount of data storage, the size of arrays, and the total size of executable programs are limited only by the amount of process virtual address space available, as determined at VAX/VMS system generation.

E.2 Run-Time Diagnostic Messages

Errors that occur during execution of your FORTRAN program are reported by diagnostic messages from the Run-Time Library. These messages may result from hardware conditions, file system errors, errors detected by RMS, errors that occur during transfer of data between the program and an internal record, computations that cause overflow or underflow, incorrect calls to the Run-Time Library, problems in array descriptions, and conditions detected by the operating system. Refer to the *VAX/VMS Run-Time Library Routines Reference Manual* for more information.

E.2.1 Run-Time Library Diagnostic Message Presentation

Run-Time Library diagnostic messages are usually sent either to your terminal (interactive mode) or to the log file (batch mode).

E.2.2 Run-Time Library Diagnostic Messages

Descriptions of Run-Time Library diagnostic messages can be found in Table E-4.

Table E-4 lists each Run-Time diagnostic message in alphabetical order according to its unique 6- to 9-character name. For organizational purposes, the message prefixes FOR\$, SS\$, and MTH\$ are not shown in this table. (Note: Refer to Table 18-1 for a presentation of the messages in order of their error numbers.)

Table E-4: Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
ADJARRDIM	93	F,C	adjustable array dimension error Upon entry to a subprogram, the evaluation of dimensioning information detected an array in which: <ul style="list-style-type: none">• An upper-dimension bound was less than a lower-dimension bound.• The dimensions imply an array that exceeds the addressable memory.
ATTACCNON	36	F	attempt to access non-existent record One of the following conditions occurred: <ul style="list-style-type: none">• An attempt was made to READ, FIND, or DELETE a nonexistent record from a relative organization file using direct access.• An attempt was made to access beyond the end of the file with a direct access READ or FIND to a sequential organization file.• An attempt was made to read a nonexistent record from an indexed organization file with a keyed access READ statement.
BACERR	23	F	BACKSPACE error One of the following conditions occurred: <ul style="list-style-type: none">• The file was not a sequential organization file.• The file was not opened for sequential access. (A unit opened for append access may not be backspaced until a REWIND statement is executed for that unit.)• RMS detected an error condition during execution of a BACKSPACE statement.

Table E-4 (Cont.): Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
CLOERR	28	F	<p>CLOSE error</p> <p>An error condition was detected by RMS during execution of a CLOSE statement.</p>
DELERR	55	F	<p>DELETE error</p> <p>One of the following conditions occurred:</p> <ul style="list-style-type: none"> • On a direct access DELETE, the file did not have relative organization. • On a current record DELETE, the file did not have relative or indexed organization, or the file was opened for direct access. • RMS detected an error condition during execution of a DELETE statement.
DUPFILSPE	21	F	<p>duplicate file specifications</p> <p>Multiple attempts were made to specify file attributes without an intervening close operation. One of the following conditions occurred:</p> <ul style="list-style-type: none"> • A DEFINE FILE was followed by DEFINE FILE. • A DEFINE FILE was followed by an OPEN statement. • A CALL ASSIGN or CALL FDBSET was followed by an OPEN statement.
ENDDURREA	24	F	<p>end-of-file during read</p> <p>One of the following conditions occurred:</p> <ul style="list-style-type: none"> • An RMS end-of-file condition was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. • An end-of-file record written by the ENDFILE statement was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. • An attempt was made to read past the end of an internal file character string or array during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification.

Table E-4 (Cont.): Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
ENDFILERR	33	F	<p>ENDFILE error</p> <p>One of the following conditions occurred:</p> <ul style="list-style-type: none"> • The file was not a sequential organization file with variable-length records. • The file was not opened for sequential or append access. • An unformatted file did not contain segmented records. • RMS detected an error during execution of an ENDFILE statement.
ERRDURREA	39	F	<p>error during read</p> <p>RMS detected an error condition during execution of a READ statement.</p>
ERRDURWRI	38	F	<p>error during write</p> <p>RMS detected an error condition during execution of a WRITE statement.</p>
FILNAMSPE	43	F	<p>file name specification error</p> <p>A file-name specification given to OPEN, INQUIRE, or CALL ASSIGN was not acceptable to RMS.</p>
FILNOTFOU	29	F	<p>file not found</p> <p>A file with the specified name could not be found during an open operation.</p>
FINERR	57	F	<p>FIND error</p> <p>RMS detected an error condition during execution of a FIND statement.</p>
FLOOVEMAT	88	F,C	<p>floating overflow in math library</p> <p>A floating overflow condition was detected during execution of a math library procedure. The result returned was the reserved operand, -0.</p>
FLOUNDMAT	89	F,C	<p>floating underflow in math library</p> <p>A floating underflow condition was detected during execution of a math library procedure. The result returned was zero.</p>

Table E-4 (Cont.): Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
FLTDIV	73	F,C	<p>arithmetic trap, zero divide</p> <p>During a floating-point or decimal arithmetic operation, an attempt was made to divide by 0.0. If floating, the result of the operation was set to the reserved operand, -0. If decimal, the result of the operation is unpredictable.</p>
FLTDIV__F	73	F,C	<p>arithmetic fault, zero divide</p> <p>During a floating-point arithmetic operation, an attempt was made to divide by zero.</p>
FLTOVF	72	F,C	<p>arithmetic trap, floating overflow</p> <p>During an arithmetic operation a floating-point value exceeded the largest representable value for that data type. The result of the operation was set to the reserved operand, -0.</p>
FLTOVF__F	72	F,C	<p>arithmetic fault, floating overflow</p> <p>During an arithmetic operation, a floating-point value exceeded the largest representable value for that data type.</p>
FLTUND	74	F,C	<p>arithmetic trap, floating underflow</p> <p>During an arithmetic operation a floating-point value became less than the smallest representable value for that data type and was replaced with a value of zero.</p>
FLTUND__F	74	F,C	<p>arithmetic fault, floating overflow</p> <p>During an arithmetic operation a floating-point value became less than the smallest representable value for that data type.</p>
FORVARMIS	61	F,C	<p>format/variable-type mismatch</p> <p>An attempt was made either to read or write a real variable with an integer field descriptor (I or L), or to read or write an integer or logical variable with a real field descriptor (D, E, F, or G). If execution continued, the following actions occurred:</p> <ul style="list-style-type: none"> • If I or L, convert as if INTEGER*4. • If D, E, F, or G, convert as if REAL*4.

Table E-4 (Cont.): Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
INCFILORG	51	F	<p>inconsistent file organization</p> <p>One of the following conditions occurred:</p> <ul style="list-style-type: none"> • The file organization specified in an OPEN statement did not match the organization of the existing file. • The file organization of the existing file was inconsistent with the specified access mode; that is, direct access was specified with an indexed organization file, or keyed access was specified with a sequential or relative organization file.
INCKEYCHG	50	F	<p>inconsistent key change or duplicate key</p> <p>A WRITE or REWRITE to an indexed organization file caused a key field to change or be duplicated. This condition was not allowed by the attributes of the file, as established when the file was created.</p>
INCOPECLO	46	F	<p>inconsistent OPEN/CLOSE parameters</p> <p>Specifications in an OPEN or CLOSE statement were inconsistent. Some invalid combinations are:</p> <ul style="list-style-type: none"> • READONLY with STATUS='NEW' or STATUS='SCRATCH' • ACCESS='APPEND' with READONLY, STATUS='NEW', or STATUS='SCRATCH' • DISPOSE='SAVE', 'PRINT', or 'SUBMIT' with STATUS='SCRATCH' • DISPOSE='DELETE' with READONLY
INCRECLEN	37	F	<p>inconsistent record length</p> <p>One of the following conditions occurred:</p> <ul style="list-style-type: none"> • An attempt was made to create a new relative, indexed, or direct access file without specifying a record length. • An existing file was opened in which the record length did not match the record size given in an OPEN or DEFINE FILE statement.

Table E-4 (Cont.): Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
INCRECTYP	44	F	<p>inconsistent record type</p> <p>The RECORDTYPE value in an OPEN statement did not match the record type attribute of the existing file that was opened.</p>
INFFORLOO	60	F	<p>infinite format loop</p> <p>The format associated with an I/O statement that included an I/O list had no field descriptors to use in transferring those values.</p>
INPCONERR	64	F,C	<p>input conversion error</p> <p>During a formatted input operation, an invalid character was detected in an input field, or the input value overflowed the range representable in the input variable. The value of the variable was set to zero.</p>
INPRECTOO	22	F	<p>input record too long</p> <p>A record was read that exceeded the explicit or the default record length specified at OPEN (or by the default OPEN). To read the file, use an OPEN statement with a RECL value of the appropriate size.</p>
INPSTAREQ	67	F	<p>input statement requires too much data</p> <p>An unformatted READ statement attempted to read more data than existed in the record being read.</p>
INSVIRMEM	41	F	<p>insufficient virtual memory</p> <p>The FORTRAN Run-Time Library attempted to exceed its virtual page limit while dynamically allocating space.</p>
INTDIV	71	F,C	<p>arithmetic trap, integer zero divide</p> <p>During an integer arithmetic operation, an attempt was made to divide by zero. The result of the operation was set to the dividend, which is equivalent to division by one.</p>
INTOVF	70	F,C	<p>arithmetic trap, integer overflow</p> <p>During an arithmetic operation, an integer value exceeded byte, word, or longword range. The result of the operation was the correct low-order part.</p>

Table E-4 (Cont.): Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
INVARGFOR	48	F	<p>invalid argument to FORTRAN Run-Time Library</p> <p>One of the following conditions occurred:</p> <ul style="list-style-type: none"> • An invalid argument was given to a PDP-11 FORTRAN compatibility subroutine such as ERRSET. • The FORTRAN compiler passed an invalid coded argument to the Run-Time Library. This can occur if the compiler is newer than the Run-Time Library in use.
INVARGMAT	81	F	<p>invalid argument to math library</p> <p>One of the mathematical procedures detected an invalid argument value.</p>
INVKEYSPE	49	F	<p>invalid key specification</p> <p>A key specification in an OPEN statement or in a keyed access READ statement was invalid. For example, the key length may have been zero or greater than 255 bytes, or the key length may not conform to the key specification of the existing file.</p>
INVLOGUNI	32	F	<p>invalid logical unit number</p> <p>A logical unit number greater than 99 or less than zero was used in an I/O statement.</p>
INVREFVAR	19	F	<p>invalid reference to variable in NAMELIST input</p> <p>The variable in error is shown as "varname" in the message text. One of the following conditions occurred:</p> <ul style="list-style-type: none"> • The variable was not a member of the namelist group. • An attempt was made to subscript the scalar variable. • A subscript of the array variable was out-of-bounds. • There were too many or too few subscripts for the variable.

Table E-4 (Cont.): Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
INVREFVAR (Cont.)	19	F	<ul style="list-style-type: none"> • An attempt was made to specify a substring of a noncharacter variable or array name. • A substring specifier of the character variable is out-of-bounds. • A subscript or substring specifier of the variable was not an integer constant. • An attempt was made to substring the unsubscripted array variable.
KEYVALERR	45	F	<p>keyword value error in OPEN statement</p> <p>An OPEN or CLOSE statement keyword requiring a value had an improper value.</p>
LISIO__SYN	59	F,C	<p>list-directed I/O syntax error</p> <p>The data in a list-directed input record had an invalid format, or the type of the constant was incompatible with the corresponding variable. The value of the variable was unchanged.</p>
LOGZERNEG	83	F,C	<p>logarithm of zero or negative value</p> <p>An attempt was made to take the logarithm of zero or of a negative number. The result returned was the reserved operand, -0.</p>
MIXFILACC	31	F	<p>mixed file access modes</p> <p>One of the following conditions occurred:</p> <ul style="list-style-type: none"> • An attempt was made to use both formatted and unformatted operations on the same unit. • An attempt was made to use an invalid combination of access modes on a unit, such as direct and sequential. The only valid combination is sequential and keyed on a unit opened with ACCESS='KEYED'. • An attempt was made to execute a FORTRAN I/O statement on a logical unit that was opened by a language other than FORTRAN.
NO__CURREC	53	F	<p>no current record</p> <p>A REWRITE or current record DELETE was attempted when no current record was defined.</p>

Table E-4 (Cont.): Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
NO_SUCDEV	42	F	no such device A file-name specification included an invalid or unknown device name when an open operation was attempted.
NOTFORSPE	1	F	not a FORTRAN-specific error An error occurred in the user program or in the Run-Time Library that was not a FORTRAN-specific error.
OPEDEFREQ	26	F	OPEN or DEFINE FILE required for keyed or direct access One of the following conditions occurred: <ul style="list-style-type: none"> • A direct access READ, WRITE, FIND, or DELETE statement was attempted for a file when no DEFINE FILE or OPEN statement with ACCESS='DIRECT' was performed for that file. • A keyed access READ statement was attempted for a file when no OPEN statement with ACCESS='KEYED' was performed for that file.
OPEFAI	30	F	open failure An error was detected by RMS while attempting to open a file in an OPEN, INQUIRE, or other I/O statement. This message is used when the error condition is not one of the more common conditions for which specific error messages are provided.
OUTCONERR	63	E,C	output conversion error During a formatted output operation, the value of a particular number could not be output in the specified field length without loss of significant digits. The field is filled with asterisks.
OUTSTAOVE	66	F	output statement overflows record An output statement attempted to transfer more data than would fit in the maximum record size.

Table E-4 (Cont.): Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
RECIO_OPE	40	F	<p>recursive I/O operation</p> <p>While processing an I/O statement for a logical unit, another I/O operation on the same logical unit was attempted. One of the following conditions may have occurred:</p> <ul style="list-style-type: none"> • A function subprogram that performs I/O to the same logical unit was referenced in an expression in an I/O list or variable format expression. • An I/O statement was executed at AST level for the same logical unit. • An exception handler (or a procedure it called) executed an I/O statement in response to a signal from an I/O statement for the same logical unit.
RECNUMOUT	25	F	<p>record number outside range</p> <p>A direct access READ, WRITE, or FIND statement specified a record number outside the range specified when the file was created.</p>
REWERR	20	F	<p>REWIND error</p> <p>One of the following conditions occurred:</p> <ul style="list-style-type: none"> • The file was not a sequential organization file. • The file was not opened for sequential or append access. • RMS detected an error condition during execution of a REWIND statement.
REWRITERR	54	F	<p>REWRITE error</p> <p>RMS detected an error condition during execution of a REWRITE statement.</p>
SEGRECFOR	35	F	<p>segmented record format error</p> <p>An invalid segmented record control data word was detected in an unformatted sequential file. The file was probably either created with RECORDTYPE='FIXED' or 'VARIABLE' in effect, or was written by a language other than FORTRAN.</p>

Table E-4 (Cont.): Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
SIGLOSMAT	87	F,C	<p>significance lost in math library</p> <p>The magnitude of an argument or the magnitude of the ratio of the arguments to a math library function was so large that all significance in the result was lost. The result returned was the reserved operand, -0.</p>
SPERECLOC	52	F	<p>specified record locked</p> <p>A READ or direct access WRITE, FIND, or DELETE was attempted on a record which was locked by another user.</p>
SQUROONEG	84	F,C	<p>square root of negative value</p> <p>An argument required the evaluation of the square root of a negative value. The result returned was the reserved operand, -0.</p>
SUBRNG	77	F,C	<p>trap, subscript out of range</p> <p>An array reference was detected outside the declared array bounds.</p>
SYNERRFOR	62	F	<p>syntax error in format</p> <p>A syntax error was encountered while the Run-Time Library was processing a format stored in an array or character variable.</p>
SYNERRNAM	17	F	<p>syntax error in NAMELIST input "text"</p> <p>The syntax of input to a namelist-directed READ statement was incorrect. The part of the record in which the error was detected is shown as "text" in the message text.</p>
TOOMANREC	27	F	<p>too many records in I/O statement</p> <p>One of the following conditions occurred:</p> <ul style="list-style-type: none"> • An attempt was made to read or write more than one record with an ENCODE or DECODE statement. • An attempt was made to write more records than existed.

Table E-4 (Cont.): Run-Time Diagnostic Messages

Condition Symbol	Err No	Sev	Message Text
TOOMANVAL	18	F	too many values for NAMELIST variable "varname" An attempt was made to assign too many values to a variable during a namelist-directed READ statement. The name of the variable is shown as "varname" in the message text.
UNDEXP	82	F,C	undefined exponentiation An exponentiation was attempted which is mathematically undefined, as for example, 0.**0. The result returned was the reserved operand, -0 for floating-point operations, and zero for integer operations.
UNIALROPE	34	F	unit already open A DEFINE FILE statement specified a logical unit that was already opened.
UNLERR	56	F	UNLOCK error RMS detected an error condition during execution of an UNLOCK statement.
VFEVALERR	68	F,C	variable format expression value error The value of a variable format expression was not within the range acceptable for its intended use; for example, a field width was less than or equal to zero. A value of one was assumed, except for a P edit descriptor, for which a value of zero was assumed.
WRONUMARG	80	F	wrong number of arguments An improper number of arguments was used to call a math library procedure.
WRIREAFIL	47	F	write to READONLY file A write operation was attempted to a file that was declared READONLY by the currently active OPEN.

E.3 DICTIONARY Error Messages

When an error occurs while using the Common Data Dictionary (CDD) (that is, while compiling a DICTIONARY statement), error messages will be generated from one or more of the following sources:

1. The FORTRAN compiler, which generates error messages that begin with %FORT. These messages appear in Section E.1.
2. The Common Data Dictionary, which generates error messages that begin with %CDD. These messages appear in Appendix D of the *VAX Common Data Dictionary Utilities Reference Manual*. CDDL error messages appear in Appendix C of the *VAX Common Data Dictionary Data Definition Language Reference Manual*.
3. The CRX, which generates error messages that begin with %CRX. These messages are listed in this section.

Most CRX messages are related to errors that cannot be corrected by the user. As indicated, submit an SPR to CDD or to the product that created the record description when you receive one of these messages.

The informational messages are related to problems that do not inhibit the production of an object file. They indicate, however, that your results may not be as you had anticipated.

Table E-5: CRX Error Messages

CRX Error	Message	User Action
%CRX-E-BADBASE	Field description specifies base other than 2 or 10.	Correct the description to be base 2 or 10.
%CRX-E-BADCORLEV	Record description specifies unsupported core level.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-BADDIGITS	Field description specifies improper record format.	Correct the field description to specify the proper number of digits.
%CRX-E-BADFORMAT	Record description specifies improper record format.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-BADLENGTH	Field description specifies improper length.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-BADOCURS	Dimension description improperly specifies Minimum Occurs.	Submit an SPR to CDD or to the product that created the description.

Table E-5 (Cont.): CRX Error Messages

CRX Error	Message	User Action
%CRX-E-BADOFFSET	Field description specifies improper offset.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-BADOVERLAY	Field description specifies overlay for nonoverlay field.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-BADPRTCL	Pathname does not designate a node with record protocol.	Correct the pathname.
%CRX-E-BADREFER	Field description specifies reference for nonpointer field.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-BADSCALE	Field description specifies scale greater than precision.	Correct the precision or scale specified in the field description.
%CRX-E-BADSTRIDE	Dimension description specifies improper stride.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-BADTAGVAR	Field description specifies tag for nonoverlay field.	Submit an SPR to CDD or to the product that created the description.
%CRX-I-INITVAL	Initial value in field description being ignored.	No action.
%CRX-I-LITERALS	Literal definitions in record description being ignored.	No action.
%CRX-E-MEMBADTYP	Field description specifies data type for field with members.	Submit an SPR to CDD or to the product that created the description.
%CRX-I-NOCONTIN	Improper continuation after a non-continuable condition.	Submit a FORTRAN SPR.
%CRX-E-NOCORATT	Record description does not specify core level.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-NOFORMAT	Record description does not specify record format.	Submit an SPR to CDD or to the product that created the description.

Table E-5 (Cont.): CRX Error Messages

CRX Error	Message	User Action
%CRX-E-NOLENGTH	Field description does not specify length.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-NOLOWER	Dimension description does not specify lower bound.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-NOOFFSET	Field description does not specify offset.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-NOOVERLAY	Field description does not specify overlay for overlay field.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-NOSTRIDE	Dimension description does not specify stride.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-NOTCOMPUT	Field definition specifies numeric attributes for nonnumeric data.	Submit an SPR to CDD or to the product that created the description.
%CRX-E-NOUPPER	Dimension description does not specify upper bound.	Submit an SPR to CDD or to the product that created the description.
%CRX-I-REFERENCE	Reference in overlay description being ignored.	No action.
%CRX-I-TAGVALUES	Tag values in overlay description being ignored.	No action.
%CRX-E-UNALIGNED	Field description specifies improper field alignment.	Correct the field description to specify the proper alignment.
%CRX-I-UNKFACIL	Unknown facility specified for record description extraction.	Submit a FORTRAN SPR.

Index

- ! (exclamation point)
 - comment indicator, 5-10
 - " (double quotation marks)
 - octal notation for integer constants, A-1, A-6
 - \$ (dollar sign)
 - delimiter for namelist record, 11-30
 - edit descriptor, 12-24 to 12-25
 - * (asterisk)
 - comment line indicator, 5-10
 - format specifier
 - in list-directed I/O, 11-16, 11-18
 - multiplication operator, 6-33 to 6-34, 6-40, 17-34
 - upper bound of array, 6-18
 - ** (double asterisk)
 - exponentiation operator, 6-33 to 6-34, 6-40
 - + (plus)
 - addition or unary plus operator, 6-33 to 6-34, 6-40, 17-34
 - (minus)
 - subtraction or unary minus operator, 6-33 to 6-34, 6-40, 17-34
 - (period)
 - current location indicator (Debug), 17-28
 - operand contents specifier (Debug), 17-34
 - / (slash)
 - division operator, 6-33 to 6-34, 6-40, 17-34
 - record terminators
 - in FORMAT statements, 12-1
 - // (double slash)
 - concatenation operator, 6-37, 6-40, 16-2
 - : (colon)
 - edit descriptor, 12-25
 - ? (question mark)
 - namelist prompt, 11-32
 - @ (AT sign)
 - execute procedure (DCL) 1-27 to 1-28
 - operand contents specifier (Debug), 17-34 (circumflex)
 - previous location indicator (Debug), 17-28
- ## A
- A field descriptor, 12-18 to 12-20
 - ACCEPT statement, 11-52
 - ACCESS
 - INQUIRE statement specifier, 13-19, 13-20
 - OPEN statement keyword, 13-3, 13-6
 - Access keys
 - for indexed files
 - specified in OPEN statement, 13-12 to 13-13
 - Access modes
 - direct, 11-15
 - OPEN statement keywords, 13-2
 - keyed, 11-15
 - sequential, 11-14 to 11-15
 - Access, shared
 - SHARED keyword
 - on OPEN statement, 13-4, 13-16
 - Actual and dummy arguments
 - see arguments, actual and dummy
 - Addition operator (+), 6-33 to 6-34, 6-40
 - Address correlation table
 - effect of /DEBUG, 3-8
 - Addresses
 - specifying program addresses (Debug), 17-29
 - virtual memory locations
 - defining symbolically (Debug), 17-31

- Adjustable arrays, 6-21, 10-3 to 10-4
- Aggregate assignment statement, 7-5 to 7-6
- Aggregate field reference
 - see record and field references
- Aggregate reference
 - definition and examples, 6-31 to 6-32
- Allocation
 - file storage allocation
 - OPEN statement keywords, 13-2
- Allocation listing, memory
 - linker output, 4-2, 4-4 to 4-5
- ALLOCATE command (Debug), 17-27
- Alphanumeric data
 - using character data type to
 - manipulate, 16-1 to 16-10
 - I/O example, 16-7 to 16-10
 - see also character data type
- Alternate key fields
 - definition, 15-2
 - discussion of use, 15-2 to 15-5
- Alternate return arguments, 10-6 to 10-7
- .AND.
 - see logical operators
- ANSI standards
 - flagging extensions
 - SYNTAX parameter (/SHOW), 3-14
 - VAX FORTRAN extensions, v, 5-1
 - see also FORTRAN-66, FORTRAN-77
- 'APPEND'
 - OPEN statement keyword value, 13-3 to 13-6
- APPEND command (DCL), 1-22 to 1-23
- Arguments, actual and dummy
 - general description, 10-2 to 10-8
 - overview, 10-2
 - adjustable arrays, 10-3 to 10-4
 - alternate return arguments, 10-6 to 10-7
 - assumed-size arrays, 10-4 to 10-5
 - character arrays, 10-5 to 10-6
 - defaults for arguments passing, 10-7 to 10-8
 - Hollerith and character constants, 10-6
 - passed-length character arguments, 10-5 to 10-6, 16-4 to 16-5
 - associating variables with, 6-16
 - bit function arguments, D-43 to D-45
 - use of aggregate field references, 6-29
 - use of built-in functions
 - Arguments, actual and dummy, (Cont.)
 - argument list functions (%VAL, %REF, %DESCR), 10-7 to 10-8
 - %LOC function, 10-8
 - use of external procedure names, 8-16 to 8-17
 - use of intrinsic function names, 8-18
 - Argument list built-in functions
 - %VAL, %REF, %DESCR, 10-7 to 10-8
 - use with Debug, 17-19
- Arithmetic assignment statement, 7-1 to 7-3
- Arithmetic expressions, 6-33 to 6-37
 - compile-time arithmetic expressions
 - use in PARAMETER statements, 8-21, 8-22
 - use in relational expressions, 6-38, 6-39
- Arithmetic IF statement, 9-13
- Arithmetic operators
 - in expressions, 6-33 to 6-34, D-1
- Arithmetic overflow
 - check options at compilation, 3-6
- Arrays
 - definition of, 6-17
 - general description, 6-17
 - adjustable, 6-21, 10-3 to 10-4
 - array names, references to
 - in FORTRAN statements, 6-21
 - assigning values to
 - with DATA statements, 8-4 to 8-6
 - assumed-size, 6-21, 10-4 to 10-5
 - character substrings
 - see character substrings
 - contrasted with records, 6-23
 - data typing of, 6-21
 - declarators, 6-18 to 6-19
 - dimensions
 - see dimensions, DIMENSION statement
 - dummy, 10-4 to 10-5
 - elements, 6-19 to 6-20
 - terminology used to describe, 6-31 to 6-32
 - making arrays equivalent, 8-11 to 8-16
 - initializing character arrays, 8-4 to 8-6, 16-4
 - subscripts, 6-19
 - unsubscripted arrays
 - statements used in, 6-21
 - use in structure declaration blocks, 6-24

Arrays, (Cont.)
 terminology used to describe,
 6-31 to 6-32

Array name reference
 definition and examples, 6-31 to 6-32

Array references
 check options at compilation, 3-6

ASCII character set, B-2

ASCII control characters
 assigning to character data entities
 with DATA statements, 8-5, 8-6

ASSIGN command (DCL), 1-18, 1-34, 11-9

ASSIGN statement, 7-6 to 7-7

Assigned GO TO statement
 general description, 9-12
 statement label references, establishing,
 7-6
 see also ASSIGN statement

Assignment operations (DCL)
 assigning character values to symbols,
 1-29
 assigning numeric values to symbols,
 1-29 to 1-31

Assignment statements
 aggregate, 7-5 to 7-6
 arithmetic, 7-1 to 7-3
 character, 7-4 to 7-5
 logical, 7-3 to 7-4

ASSOCIATEVARIABLE
 OPEN statement keyword, 13-3, 13-6

Assumed-size arrays, 6-21, 10-4 to 10-5

Asterisk (*)
 comment line indicator, 5-10
 format specifier
 in list-directed I/O, 11-16,
 11-18
 multiplications operator, 6-33 to
 6-34, 6-40, 17-34
 upper bound of array, 6-18

AT sign (@)
 execute procedure (DCL), 1-27 to 1-28
 operand contents specifier (Debug), 17-34

ATTACH command (Debug), 17-4

B

BACKSPACE statement
 general description, 13-27
 see also REWIND statement

Bit field transfers
 MVBITS subroutine, D-42 to D-43

Bit functions
 general information about, D-43
 to D-45

BLANK
 INQUIRE statement specifier, 13-19,
 13-20
 OPEN statement keyword, 13-3, 13-6

Blank characters
 see space characters

Blank common blocks, 8-3

Blank control editing
 BN and BZ edit descriptors, 12-5

Blank lines
 treated as comment lines, 5-10

Blocks
 OPEN keywords affecting, 13-2

Block data program unit, 8-2

BLOCK DATA statement, 8-2 to 8-3

Block data subprograms
 forcing linker to search libraries,
 8-17

Block IF constructs, 9-14 to 9-20

BLOCKSIZE
 OPEN statement keyword, 13-3, 13-7

BN
 edit descriptor, 12-5

Breakpoints
 setting in debug sessions, 17-20 to 17-23

BUFFERCOUNT
 OPEN statement keyword, 13-7

Built-in functions
 argument list functions
 %VAL, %REF, %DESCR, 10-7 to 10-8
 %LOC function, 10-8

BYTE data type
 relationship to LOGICAL*1, 8-7
 see also LOGICAL

BZ
 edit descriptor, 12-5

C

C comment indicator, 5-10

Calendar dates
 subroutines for calculating
 DATE and IDATE, D-39

CALL command (Debug), 17-19

CALL statement, 9-2
 use with ENTRY statement, 10-18, 10-19
 use with EXTERNAL statement, 8-17
 use with INTRINSIC statement, 8-18, 8-19
 use with RETURN statement, 9-21, 9-22
 use with SUBROUTINE statement, 10-14, 10-15, 10-16

CANCEL commands (Debug)
 CANCEL BREAK, 17-22
 CANCEL DISPLAY, 17-43
 CANCEL MODULE, 17-26
 CANCEL TRACE, 17-22
 CANCEL WATCH, 17-23
 CANCEL WINDOW, 17-40

CARRIAGECONTROL
 INQUIRE statement keyword, 13-19, 13-20
 OPEN statement keyword, 13-3, 13-8

Carriage control editing, 12-24 to 12-25

Carriage control characters, 12-25 to 12-26

CDD, 3-19 to 3-23
 CDD records
 creating CDD records, 3-22
 Gfloating/Dfloating caution, 3-23
 including in source listing, 3-13, 3-20, 3-27
 see also DICTIONARY statement, /DICTIONARY qualifier

CDD\$TOP, 3-27

CDDL, 3-19, 3-20
 data types supported, 3-22, 3-23

CDDV, 3-20

Cells
 in relative organization files, 11-10, 11-11

Change mode editing (EDT)
 see character mode editing

CHAR function, 10-25, 16-5

CHARACTER
 constants
 as actual arguments, 10-6
 general description, 6-13
 general discussion of use, 16-1 to 16-10
 I/O examples, 16-7 to 16-10
 data type
 definition, 6-1
 representation in memory, C-10

CHARACTER, data type, (Cont.)
 storage requirement, 6-3
 see also arrays, constants, data types, variables

CHARACTER*n
 see CHARACTER

Character, continuation, 5-11

Characters
 flagging lowercase in output (/STANDARD), 3-14
 in character and Hollerith constants, 5-7
 supported by VAX FORTRAN, 5-6
 see also character sets

Character arguments
 passed length, 6-17, 10-5 to 10-6, 16-4 to 16-5

Character assignment statement, 7-4 to 7-5

Character editing (A,H), 12-18 to 12-21
 character constants, 12-20, 12-21

Character expressions, 6-37
 compile-time character expressions
 use in PARAMETER statements, 8-21, 8-22
 operators, 6-37, D-1
 use in relational expressions, 6-38, 6-39

Character comparison library functions
 LEN, INDEX, ICHAR, CHAR, 10-24 to 10-26, 16-5 to 16-6
 see also lexical comparison functions

CHARACTER FUNCTION statement, 10-12

Character mode editing (EDT), 2-1, 2-6 to 2-12
 deleting text, 2-10 to 2-11
 inserting text, 2-10
 moving text, 2-11 to 2-12
 positioning cursor, 2-7 to 2-10
 undeleting text, 2-10 to 2-11

Character operators, 6-37, D-1

Character scalar memory reference
 see scalar memory reference

Character sets
 ASCII, B-2
 assigning ASCII characters to data entities, 8-5, 8-6
 FORTRAN, B-1
 Radix-50, B-3 to B-4

Character statement
 see character type declaration statement

Character substrings
 definition, 6-4
 concatenating, 16-2
 establishing equivalence among, 8-13 to 8-16
 use in variables and arrays, 6-22 to 6-23
 terminology used to describe, 6-31 to 6-32

Character type declaration statement
 general description, 8-8 to 8-9

Character variables
 initializing, 6-4

/CHECK qualifier, 3-6

Circumflex ()
 previous location indicator (Debug), 17-28

CLOSE statement
 general description, 13-18 to 13-19

Coding form, 5-8

Coding requirements
 see source code

Colon (:)
 edit descriptor, 12-25

Column(s)
 one
 comment indicator, 5-10
 one through five
 statement label field, 5-10
 six
 continuation indicator, 5-11
 seven through 72 (optionally, to 132)
 statement field, 5-2, 5-11
 73 - 80
 sequence number field, 5-11

Command procedures
 see DCL command procedures,
 debugger command procedures

Command qualifiers
 rules for specifying, 1-3

Comment line indicators
 D in column 1, 3-9, 5-10
 general description, 5-5

Common blocks
 COMMON and EQUIVALENCE
 interaction, 8-16
 establishing order of contents, 8-3 to 8-4
 initializing values in, 8-2 to 8-3

Common block names
 use in COMMON statement, 8-3

Common Data Dictionary
 see CDD

COMMON statement
 general description, 8-3 to 8-4
 interaction with EQUIVALENCE, 8-16
 establishing arrays with, 6-17
 establishing variables with, 6-16
 use of unsubscripted arrays with, 6-21

Compatibility
 additional language elements for,
 A-1 to A-8

Compilation options
 options affecting output
 contents of source listing file
 (/SHOW), 3-13
 debugging information (/DEBUG,
 /OPTIMIZE), 3-8, 3-12, 17-2, 17-3
 messages (/STANDARD and
 /WARNINGS), 3-14 to 3-15
 object code listing (/MACHINECODE),
 3-12
 object file name (/OBJECT), 3-12
 source listing (/LIST), 3-11
 options affecting processing
 checking overflow, bounds, underflow
 (/CHECK), 3-6
 continuation line limits
 (/CONTINUATION), 3-7
 D in column 1 (/DLINES), 3-9
 FDML preprocessing (/DML), 3-9
 FORTRAN-77 or FORTRAN-66 (/F77),
 3-10
 Gfloating vs. Dfloating (/GFLOATING),
 3-10
 INTEGER and LOGICAL defaults (/I4),
 3-11
 optimization (/OPTIMIZE), 3-12
 source line length (/EXTENDSOURCE),
 3-9
 text library files (/LIBRARY), 3-11, 3-18
 overriding options on FORTRAN
 command
 OPTIONS statement, 3-24, 3-26

Compilation summary listing, 3-35

Compiler
 coding restrictions/limits
 summary of, E-27 to E-28
 default file names, 11-7
 diagnostic messages issued by

- Compiler, (Cont.)
 - compiler-fatal diagnostic messages, E-26 to E-27
 - source program diagnostic messages, E-1 to E-26
 - functions, 3-1 to 3-2
 - input to linker, 3-1 to 3-2
- Compiler output listing
 - see output listing
- Compile-time constant expression
 - arithmetic, character, and logical, 8-21
- Completion status values
 - returning to command procs, 4-8
- Complex data editing, 12-15 to 12-16
- COMPLEX
 - see COMPLEX*8
- COMPLEX*8
 - constants, 6-9 to 6-10
 - data type
 - definition, 6-1
 - representation in memory, C-6
 - storage requirement, 6-3
 - see also arrays, constants, data types, variables
- COMPLEX*16
 - constants, 6-10
 - data type
 - Gfloating vs. Dfloating, 3-10, 6-3
 - representation in memory, C-6 to C-8
 - storage requirement, 6-3
 - see also arrays, constants, data types, variables
- Computed GOTO statement, 9-11
- Concatenation operator (//), 6-37, 6-40, 16-2
- Condition handlers, 18-1, 18-2
- Condition symbols, FORTRAN
 - summary of, 18-3 to 18-5
- Connections, logical
 - to logical I/O units
 - explicitly by means of OPEN, 13-17
 - implicitly by system default, 11-7 to 11-8
- Constants
 - definition of, 6-4
 - assigning symbolic names
 - by means of PARAMETER statement, 8-21 to 8-22
 - character, 6-13
- Constants, (Cont.)
 - complex, 6-9 to 6-10
 - hexadecimal, 6-11 to 6-12
 - Hollerith, 6-14
 - integer, 6-5
 - logical, 6-13
 - octal, 6-11 to 6-12
 - Radix-50, B-3
 - real 6-5 to 6-9
 - terminology used to describe, 6-31 to 6-32
- Continuation character, 5-11
- Continuation lines
 - DCL commands, 1-7
 - FORTRAN source code
 - debug source statements, use in, 5-10
 - indicator in source code, 5-11
 - source program limits, how to modify, 3-7
 - statement labels, affect on, 5-10
 - /CONTINUATION qualifier, 3-7
- CONTINUE command (DCL), 4-7, 17-5
- CONTINUE statement, 9-3
- Control statements, FORTRAN, 9-1 to 9-23
 - see also CALL, RETURN
- Conversion
 - of data types
 - in arithmetic assignment statements, 7-3
 - in DATA statements, 8-5 to 8-6
 - with FORMAT statements, 12-1
- COPY command (DCL), 1-22
- CREATE /DIRECTORY command (DCL), 1-15 to 1-16
- Cross-reference information
 - compiling
 - /CROSSREFERENCE qualifier, 3-7
 - /SHOW qualifier (SINGLE parameter), 3-13
 - general description, 3-23
 - linking
 - /CROSSREFERENCE qualifier, 4-2
- <CTRL/C> and <CTRL/Y>
 - interrupting interactive program execution, 4-7 to 4-8
 - use of <CTRL/y> during debug session, 17-4 to 17-5
- <CTRL/z>
 - terminating a debug session, 17-5
- Cursor positioning (EDT), 2-7 to 2-10

D

D debugging statement indicator
use in column 1, 3-9, 5-10

field descriptor, 12-13
in complex data editing, 12-15

Data

as stored in memory
by VAX FORTRAN, C-1 to C-11
editing
with FORMAT statements, 12-1
examining and manipulating (Debug),
17-32 to 17-38
passing to programs, 1-33
retaining after END or RETURN, 8-24
to 8-25
see also data items, data types

Data items

definition, 6-4
printing warnings if unused, 3-15
terminology used to describe, 6-31 to 6-32
see also arrays, character substrings,
constants, records, variables

Data Manipulation Language, 3-9

DATA statement

general description, 8-4 to 8-6
use of unsubscripted arrays with, 6-21, 8-5
use to define arrays and elements, 6-17

Data transfer

see I/O

Data types

conversion rules
for arithmetic assignment statements,
7-3
for DATA statements, 8-5 to 8-6
declaration within structures, 8-28 to 8-30
default data types
of undeclared symbolic names, 8-17
definition of different types, 6-1 to 6-2
expressions
establishing data types of, 6-35 to 6-37
FORTRAN-to-CDD mapping, 3-22 to 3-23
INTEGER and LOGICAL
setting default lengths, 3-11
length specifiers, 6-2
method of specifying
arrays, 6-21
constants, 6-4 to 6-15
variables, 6-15, 6-16 to 6-17

Data types, method of specifying, (Cont.)
rank in arithmetic expressions, 6-35
specifying during debug sessions, 17-35
to 17-36

see also BYTE, CHARACTER,
INTEGER,
LOGICAL, REAL

Data type declaration statement

general descriptions
character type declarations, 8-8 to 8-9
numeric type declarations, 8-7 to 8-8
use of unsubscripted arrays with, 6-21
use to establish arrays, 6-17
use to establish variables, 6-15,
6-16 to 6-17

DATE subroutine, D-39

Dates, calendar

subroutines for calculating
DATE and IDATE, D-39

DBG\$INIT, 17-9

DBG\$INPUT, 17-9

DCL commands

continuation indicator, 1-7
issuing during debug sessions, 17-4
rules and options, 1-7 to 1-9

DCL command procedures, 1-27 to 1-38

controlling execution 1-35 to 1-36
creation of symbols, 1-28 to 1-32
error handling, 1-36 to 1-37
executing, 1-28, 1-37 to 1-38
login command file, 1-38
passing data to programs, 1-34 to 1-35
passing parameters, 1-32 to 1-33

DEASSIGN command (DCL), 1-20

DEBUG command (DCL), 17-4, 17-5

Debugger

command summary, 17-11 to 17-17
command entry in keypad mode
keypad command layout, 17-7
controlling program execution, 17-17
to 17-23
using logical control commands, 17-25
defining special function keys, 17-6,
17-7 to 17-8
displaying source lines, 17-23 to 17-24
examining and manipulating data, 17-32
to 17-38
debugger operators, 17-34
examples sessions, 17-43 to 17-48

- Debugger, (Cont.)
 - invoking, interrupting, terminating sessions, 17-4 to 17-5
 - referencing symbolic names and programs locations, 17-25 to 17-32
 - use of prefixes and scope, 17-29 to 17-31
 - using screen displays, 17-38 to 17-43
 - see also /DEBUG qualifier
- Debugger command procedures, 17-8 to 17-9, 17-10
- Debugger symbol table (DST), 17-26
- Debugging statements
 - in source code, 3-9, 5-10
- /DEBUG qualifier
 - FORTRAN command, 3-8
 - LINK command, 4-2, 4-3
 - overview/summary, 4-9 to 4-11
 - RUN command, 4-7
- Declarators, array, 6-18
- DECODE statement, A-1 to A-3
- Defaults
 - argument passing defaults, 10-7 to 10-8
 - data type defaults
 - of undeclared symbolic names, 8-17
 - field descriptors vs. I/O list elements, 12-21
 - logical I/O unit names, 11-7 to 11-8
 - predefined system logical names, 1-19
- DEFAULTFILE
 - INQUIRE statement keyword, 13-19
 - OPEN statement keyword, 13-3, 13-18
- DEFINE command (DCL), 1-18
- DEFINE commands (Debug)
 - DEFINE, 17-31
 - DEFINE/KEY, 17-7 to 17-8
- DEFINE FILE statement, A-1, A-3 to A-4
- 'DELETE'
 - file description, 13-18
- DELETE command (DCL), 1-23
- DELETE statement
 - general description, 13-28 to 13-29
- DELETE/KEY command (Debug), 17-7, 17-8
- Deleting files
 - DELETE command (DCL), 1-23
 - PURGE command (DCL), 1-23 to 1-24
- DEPOSIT command (Debug), 17-28, 17-32, 17-33, 17-35, 17-38
- depositing complex values, 17-38
- %DESCR built-in function, 10-7 to 10-8
 - use with Debug, 17-9
- Descriptors
 - see field and edit descriptors
- Devices
 - default, 1-13
 - defining logical names for, 1-18
 - identifying in file specs, 1-11
- Dfloating implementations
 - effect of /GFLOATING qualifier, 3-10
 - CDD caution, 3-23
 - representation in memory, C-2
 - COMPLEX*16, C-6 to C-7
 - REAL*8, C-3
 - with COMPLEX*16 data type, 6-10
 - with REAL*8 data type, 6-3, 6-7 to 6-8
- Diagnostic messages
 - see messages
- Dictionary Management Utility, 3-20
- DICTIONARY parameter (/SHOW), 3-13
- DICTIONARY statement
 - general discussion, 3-20 to 3-21
- Dimensions
 - array declarators, 6-18
 - array limits, 6-17
 - see also DIMENSION statement
- DIMENSION statement
 - general description, 8-9 to 8-10
 - use to establish arrays, 6-17
- 'DIRECT'
 - OPEN statement keyword value, 13-3, 13-6
- DIRECT
 - INQUIRE statement specifier, 13-19, 13-21
- Direct access mode, 11-15
 - OPEN statement keywords, 13-2
 - see also relative organization files
- Direct access FIND statements, A-1, A-4 to A-5
- Direct access READ statements, 11-35 to 11-36
- Direct access WRITE statements, 11-46 to 11-47
- Directories
 - changing a file's directory, 1-22
 - default, 1-13
 - description, 1-14 to 1-17
 - defining logical names for, 1-17 to 1-18

Directories, (Cont.)
 displaying files in, 1-24
 specifying in file specs, 1-11
 DIRECTORY command (DCL), 1-24
 /PROTECTION qualifier, 1-26
 DISP
 CLOSE statement keyword, 13-18
 DISPLAY command (Debug), 17-43
 Display screens (Debug), 17-39 to 17-43
 Displaying file contents
 TYPE and PRINT commands (DCL), 1-26
 to 1-27
 TYPE and <RET> commands (EDT),
 2-17 to 2-18
 DISPOSE
 CLOSE statement keyword, 13-18
 OPEN statement keyword, 13-3, 13-9
 Division operator (/), 6-33 to 6-34, 6-40
 /DLINES qualifier, 3-9, 5-10
 /DML qualifier, 3-9
 DMU, 3-20, 3-21
 Dollar sign (\$)
 delimiter for namelist record, 11-30
 edit descriptor, 12-24 to 12-25
 DO loops
 see DO statements
 DO statements, 9-3 to 9-10
 DOUBLE COMPLEX
 see COMPLEX*16 data type
 DOUBLE PRECISION
 see REAL*8 data type
 DST (debugger symbol table), 17-26
 Dummy and actual arguments
 see subprogram arguments

E

E field descriptor, 12-11 to 12-12
 in complex data editing, 12-15
 Edit and field descriptors
 summary, 12-30
 see also FORMAT statements
 EDIT /EDT command (DCL), 2-3 to 2-4
 Editor, text
 see EDT text editor
 EDTINI.EDT file, 2-25 to 2-26
 EDT text editor, 2-1 to 2-26
 changing editing modes, 2-5

EDT text editor, (Cont.)
 character mode editing
 see character mode editing
 command summary (line editing), 2-12 to
 2-14
 editing modes
 see character mode editing, line mode
 editing
 EDTINI.EDT file, 2-25 to 2-26
 HELP facilities, 2-2 to 2-3
 invoking EDT, 2-3 to 2-4
 keypad editing
 see character mode editing
 range specifications (line editing), 2-14 to
 2-17
 setting up tabs, 2-23 to 2-24
 startup command files, 2-25 to 2-26
 terminating EDT, 2-4 to 2-5
 ELSE statement
 block IF constructs, 9-14 to 9-20
 ELSE IF THEN statement
 block IF constructs, 9-14 to 9-20
 ENCODE statement, A-1 to A-3
 END statement
 affect on program execution, 4-7
 general description, 9-10
 effecting a return from subprograms, 9-10
 when not to use, 4-8
 with BLOCK DATA statement, 8-2
 with FUNCTION statement, 10-11
 with SUBROUTINE statement, 10-14
 see also END DO, ENDFILE, ENDF,
 END MAP, END STRUCTURE, END
 UNION
 END DO statement, 9-9 to 9-10
 ENDFILE statement, 13-27 to 13-28
 END IF statement
 block IF constructs, 9-14 to 9-20
 END MAP statement
 see MAP statement, map declaration
 End-of-file condition
 reporting with IOSTAT value, 11-21
 transferring control with END specifier,
 11-21 to 11-22
 End-of-file record
 ENDFILE statement, 13-27 to 13-28
 END specifier
 in I/O statements, 11-21 to 11-22, 18-1,
 18-6

- END STRUCTURE statement
 - see STRUCTURE statement, structure declaration block
- END UNION statement
 - see UNION statement, union declaration
- Entry names
 - rules applying to, 10-16, 10-17
- ENTRY statement, 10-16 to 10-19
 - use with CALL statement, 9-2
 - use with FUNCTION statement, 10-11
 - use with SUBROUTINE statement, 10-15
- .EQ.
 - see relational operators
- EQUIVALENCE statement
 - associating variables with, 6-16
 - contrasted with union declaration, 8-32
 - general description, 8-10 to 8-16
 - interaction with COMMON, 8-16
 - use of unsubscripted arrays with, 6-16
- .EQV.
 - see logical operators
- ERR
 - I/O statement specifier, 11-21 to 11-22, 18-1, 18-6
 - BACKSPACE statement keyword, 13-27
 - CLOSE statement keyword, 13-18
 - DELETE statement keyword, 13-28, 13-29
 - ENDFILE statement keyword, 13-27, 13-28
 - INQUIRE statement specifier, 13-19, 13-21
 - OPEN statement keyword, 13-3, 13-9
 - REWIND statement keyword, 13-26
 - UNLOCK statement keyword, 13-30
- Error handling
 - condition handlers, 18-1, 18-2
 - DCL command proc errors, 1-36 to 1-37
 - processing performed by RTL, 18-1 to 18-7
 - subroutine for obtaining error info
 - ERRSNS subroutine, D-40
 - summary of FORTRAN run-time errors, 18-3 to 18-5
 - user controls in I/O statements
 - ERR, END, and IOSTAT specifiers, 11-21 to 11-22, 18-1
- Error codes
 - \$STATUS and \$SEVERITY symbols, 1-36
- Error condition
 - reporting with IOSTAT specifier, 11-21
 - transferring control with END specifier, 11-21, 11-22
- Error messages
 - see messages
- Error-related command qualifiers
 - summary, 4-9
- Error traceback mechanism
 - see traceback mechanism
- ERRSET subroutine
 - error table maintained by, 18-2
- ERRSNS subroutine, D-40
- ERRTST subroutine
 - error table maintained by, 18-2
- EVALUATE command (Debug), 17-32, 17-33, 17-34
- EXAMINE command (Debug), 17-28, 17-32, 17-33, 17-34, 17-38
 - data type qualifiers, 17-35 to 17-36
 - radix qualifier, 17-37
- Exception condition
 - common when using indexed files, 15-7 to 15-8
- Exclamation point (!)
 - comment indicator, 5-10
- /EXECUTABLE
 - LINK command option, 4-2, 4-3 to 4-4
- Executable statements
 - definition and list of, 5-2
- Execution, program, 4-6 to 4-11
 - controlling during debug session, 17-17 to 17-23
 - interrupting
 - <CTRL/C> and <CTRL/Y>, 4-7 to 4-8
 - temporarily suspending (PAUSE), 9-20 to 9-21
 - terminating
 - EXIT, D-40
 - STOP, 9-23
- EXIST
 - INQUIRE statement specifier, 13-19, 13-21
- EXIT command (DCL), 1-35 to 1-36
- EXIT system subroutine, 4-8, D-40
- Explicit formatting
 - I/O statement specifier, 11-17 to 11-18
- Exponents
 - in REAL*4 constants, 6-6
 - in REAL*8 constants, 6-7
 - in REAL*16 constants, 6-8
- Exponentiation operator (**), 6-33 to 6-34, 6-4
- Expressions
 - definition of, 6-32

Expressions, (Cont.)
 arithmetic, 6-33 to 6-37
 data type ranking, 6-35
 operator precedence, 6-33 to 6-34
 rules governing typing of, 6-35 to 6-37
 character, 6-37
 compile-time constant expressions
 arithmetic, character, and logical, 8-21
 expression operators
 summary of, D-1 to D-2
 logical, 6-39 to 6-40
 operator precedence, 6-40
 relational, 6-38 to 6-39
 terminology used to describe, 6-31
 Expressions, variable FORMAT, 12-4 to 12-5
 Extended ranges, DO LOOP, 9-7
 EXTENDSIZE
 OPEN statement keyword, 13-3, 13-10
 EXTENDSOURCE qualifier, 3-9
 affect on sequence number field, 5-11
 External field separators, 12-27
 External procedures
 invoking by means of CALL, 9-2
 External procedure names
 duplicating intrinsic function names, 8-17
 use as arguments, 8-16 to 8-17
 EXTERNAL statement, 8-16 to 8-17
 /NOF77 implementation, A-6 to A-8

F

F field descriptor, 12-10 to 12-11
 in complex data editing, 12-15
 .FALSE.
 see logical constants
 FDML, 3-9
 Floating data
 representation in memory, C-2
 COMPLEX*8, C-6
 REAL*4, C-2 to C-3
 Fields
 definition in structure declarations, 8-26,
 8-27 to 8-33
 in FORTRAN source code, 5-7 to 5-11
 I/O fields
 see FORMAT statements
 in records
 see records (structured data items)

Field and edit descriptors
 summary, 12-30
 see references to individual descriptors
 see also FORMAT statements
 Field declaration
 see structure declaration blocks
 Field descriptors, default
 for I/O list elements, 12-21
 Field reference
 see record and field references
 Field separators, external, 12-27
 FILE
 INQUIRE statement keyword, 13-19
 OPEN statement keyword, 13-4, 13-10
 Files
 assigning to logical units
 summary, 11-9 to 11-10
 changing directories/devices, 1-22
 combining files at compilation, 3-24 to 3-25
 copying to another node, 1-10 to 1-11
 creating source files (EDT), 2-7
 deleting, 1-23 to 1-24
 deleting records from
 DELETE statement, 12-28 to 12-29
 displaying contents
 TYPE and PRINT commands (DCL),
 1-26 to 1-27
 TYPE and <RET> commands (EDT),
 2-17 to 2-18
 displaying file status info, 1-24
 disposition
 CLOSE statement keywords, 13-18
 INCLUDE files, 3-24 to 3-25
 processing options
 OPEN statement keywords, 13-2
 properties, inquiring about
 INQUIRE statement, 13-19 to 13-26
 protecting, 1-25 to 1-26
 record description options, I/O
 OPEN statement keywords, 13-2
 repositioning
 BACKSPACE statement, 13-27
 REWIND statement, 13-26
 searching file contents, 1-26
 search lists, use of, 1-21
 status options
 OPEN statement keywords, 13-2
 see also internal files, source files,

Files, (Cont.)

- object files, text library
- files, listing files
- File-handling commands
 - DCL commands, 1-21 to 1-27
 - FORTTRAN statements
 - BACKSPACE statement, 13-27
 - CLOSE statement, 13-18 to 13-19
 - INQUIRE statement, 13-19 to 13-26
 - OPEN statement, 13-1 to 13-18
 - REWIND statement, 13-26
- File name
 - defining logical names for, 1-17 to 1-18, 11-5 to 11-6
 - displaying, 1-24
 - specifying in file specs, 1-11
- File organizations
 - overview (sequential, relative, indexed), 11-10 to 11-11
 - see also indexed organization files, relative organization files, sequential organization files
- File sharing
 - SHARED keyword (OPEN statement), 13-4, 13-16
- File specifications
 - defaults, 1-13
 - OPEN statement keywords, 11-8 to 11-9, 13-2
 - field descriptions, 1-10 to 1-13
 - use of wildcards, 1-14
- File status
 - CLOSE statement keywords, 13-18
 - OPEN statement keywords
 - DISPOSE, 13-3, 13-9
 - STATUS or TYPE, 13-4, 13-16 to 13-17
- File storage allocation
 - OPEN statement keywords, 13-2
- File type
 - default, 1-13
 - specifying in file specs, 1-11
- File version number
 - specifying in file specs, 1-11 to 1-12
- %FILL, 6-24
- FIND statement, A-1, A-4 to A-5
- Fixed format
 - see formats
- Fixed-length records
 - format, 11-11

Fixed-length records, (Cont.)

- RECORDTYPE keyword (OPEN statement), 13-4, 13-15 to 13-16
- Floating-point representation in memory, C-2 to C-8
- FMT format specifier
 - in I/O statements, 11-17 to 11-18
- FOR command (Debug), 17-25
- FOR\$
 - prefix for condition symbols
 - for FORTRAN run-time errors, 18-3 to 18-4
- FORM
 - INQUIRE statement specifier, 13-19, 13-22
 - OPEN statement keyword, 13-3, 13-10 to 13-11
- Formats
 - I/O formatting
 - see FMT format specifier
 - passed length, 6-3
 - run-time, 12-27 to 12-28
- Formats, coding
 - fixed format, 5-7 to 5-8
 - tab format, 5-9 to 5-10
- Format specification separators, 12-26
- Format specifier, FMT
 - see FMT format specifier
- FORMAT statements
 - description of use, 12-1
 - arithmetic expressions in, 12-4 to 12-5
 - external field separators, 12-27
 - field and edit descriptors
 - summary of, 12-30
 - counts, repeat, 12-3
 - blank control editing, (BN,BZ), 12-5
 - character editing (A,H), 12-18 to 12-21
 - character constants, 12-20, 12-21
 - integer editing (I,O,Z), 12-6 to 12-9
 - logical editing (L), 12-18
 - miscellaneous editing operations (Q,\$,:), 12-24 to 12-25
 - positional editing (X,T,TL,TR), 12-22 to 12-23
 - real editing (F,E,D,G), 12-10 to 12-16
 - scale factor editing (P), 12-16 to 12-17
 - sign control editing (SP,SS,S), 12-6
 - format expressions, variable, 12-4 to 12-5
 - format specification separators, 12-26
 - I/O lists, interaction with, 12-28 to 12-19

FORMAT statements, (Cont.)
 run-time formats, 12-27 to 12-28
 summary information about
 field and edit descriptors, 12-30
 general rules, 12-31
 input rules, 12-32
 output rules, 12-32
 syntax, 12-1 to 12-2
FORMATTED
 INQUIRE statement specifier, 13-19, 13-22
Formatted I/O statements
 ACCEPT statement, 11-52
 establishing statement label references, 7-6
 general description, 11-3
 PRINT statement, 11-53
READ statements
 direct access, 11-35, 11-37
 indexed, 11-37, 11-38
 internal, 11-39, 11-40
 sequential, 11-26, 11-27
REWRITE statement, 11-50, 11-51
TYPE statement, 11-53
WRITE statements
 direct access, 11-46, 11-47
 indexed, 11-47, 11-48
 internal, 11-49, 11-50
 sequential, 11-41, 11-42 to 11-43
FORSYSDEF.TLB, 3-19
 condition symbol values, 18-2
FORT\$LIBRARY, 3-18 to 3-19
FORTTRAN character set, B-1
FORTTRAN command (DCL), 3-2 to 3-15
 /CHECK, 3-6
 /CONTINUATIONS, 3-7
 /CROSSREFERENCE, 3-7
 /DEBUG, 3-8, 4-9, 4-10
 /DLINES, 3-9, 5-10
 /DML, 3-9
 /EXTENDSOURCE, 3-10
 affect on sequence number field, 5-11
 /F77, 3-10
 /GFLOATING, 3-10
 /I4, 3-11
 /LIBRARY, 3-11
 /LIST, 3-3, 3-4, 3-11
 /MACHINECODE, 3-12
 /OBJECT, 3-3, 3-4, 3-12
 /OPTIMIZE, 3-12
 /SHOW, 3-13
FORTTRAN command (DCL), (Cont.)
 /STANDARD, 3-14
 /WARNINGS, 3-15
 format, 3-2
 library search order, 3-18
 overriding options on FORTTRAN command
 OPTIONS statement, 3-24, 3-26 to
 3-27
 qualifier summary, 3-5
 specifying files in, 3-3 to 3-4
FORTTRAN compiler
 coding restrictions/limits
 summary of, E-27 to E-28
 default file names, 11-7
 diagnostic messages issued by
 compiler-fatal diagnostic messages,
 E-26 to E-27
 source program diagnostic messages,
 E-1 to E-26
 functions, 3-1 to 3-2
 input to linker, 3-1 to 3-2
FORTTRAN condition symbols
 for run-time errors, 18-3 to 18-5
FORTTRAN Data Manipulation Language,
 3-9
FORTTRAN data representation in memory,
 C-1 to C-11
FORTTRAN logical names
 defaults and use, 11-6 to 11-7
FORTTRAN logical unit numbers, 11-7 to 11-8
FORTTRAN statements
 assignment statements, 7-1 to 7-7
 coding restrictions/limits
 summary of, E-27 to E-28
 control statements, 9-1 to 9-23
 see also CALL, RETURN
 executable/nonexecutable, 5-2
 general description, 5-2, 5-11
 I/O statements, 11-1 to 11-53
 I/O statements, auxiliary, 13-1 to 13-30
 language summary (alphabetic), D-2 to
 D-30
 maximum line length, 3-9
 ordering requirements, 5-2 to 5-3
 specification statements, 8-1 to 8-34
 supplemental statements
 supported to maintain non-VAX
 FORTTRAN compatibility, A-1 to
 A-8

FORTRAN-66
 /NOF77 qualifier, 3-10
FORTRAN-77
 /F77 qualifier, 3-10
 /STANDARD qualifier, 3-14
 VAX FORTRAN extensions, v, 5-1
 /FULL qualifier
 LINK command option, 4-2, 4-4 to 4-5
Functions, built-in
 argument list functions
 %VAL, %REF, %DESCR, 10-7 to 10-8
 %LOC function, 10-8
Function names
 see subprograms
Function references
 general description, 10-13 to 10-13
 types of references to intrinsic functions
 specific and generic, 10-19 to 10-24
FUNCTION statement, CHARACTER, 10-12
FUNCTION statement, 10-11 to 10-14
Function subprograms, statement
 see subprograms
 /F77 qualifier, 3-10

G

G field descriptor, 12-13 to 12-15
 in complex data editing, 12-15
.GE.
 see relational operators
Generic references
 to intrinsic function names, 10-20 to 10-24
Gfloating implementations
 CDD caution, 3-23
 /GFLOATING qualifier, 3-10
 representations in memory
 COMPLEX*16, C-7 to C-8
 REAL*8, C-4
 with COMPLEX*16 data type, 6-10
 with REAL*8 data type, 6-3, 6-7 to 6-8
 /GFLOATING qualifier, 3-10
Global symbols
 assignment operations, 1-8 to 1-9, 1-29
GO command (Debug), 17-17 to 17-18
GOTO command (DCL), 1-35
GOTO statements
 general descriptions
 assigned GOTO, 9-12
 computed GOTO, 9-11
 unconditional GOTO, 9-10 to 9-11

GROUP file access, 1-25
GROUP logical name table, 1-18 to 1-19
Group repeat counts
 in FORMAT statements, 12-3
.GT.
 see relational operators

H

field descriptor, 12-20
Hexadecimal constants, 6-11 to 6-12
HELP command, 1-9 to 1-10
Hfloating data
 representation in memory, C-2
 REAL*16, C-4 to C-5
Hollerith
 data type
 definition, 6-1
 constants, 6-4
 as actual arguments, 10-6
 representation in memory, C-10 to C-11

I

I field descriptor, 12-6 to 12-7
ICHAR function, 10-25, 16-6
IDATE subroutine, D-39
IF statements
 general descriptions
 arithmetic IF, 9-13
 block IF, 9-14 to 9-20
 logical IF, 9-13 to 9-14
IF...THEN command (DCL), 1-35
IF THEN statement
 block IF constructs, 9-14 to 9-20
IF THEN ELSE command (Debug), 17-25
IMPLICIT statement
 data typing variables with, 6-5, 6-16
 effect of /WARNINGS option, 3-15, 8-18
 general description, 8-17 to 8-18
Implied-DO list
 I/O list parameter
 in DATA statements, 8-4, 8-5
 in I/O statements, 11-24 to 11-26
Implied-DO variable
 in DATA statements, 8-5
INCLUDE files
 including in output listing
 /SHOW qualifier, 3-13

- /INCLUDE qualifier (LINK), 4-3, 4-5
- INCLUDE statement, 3-15, 3-16
 - statement definition, 3-24 to 3-25
- Indefinite DO statement, pretested DO WHILE, 9-9
- Indexed DO statement, 9-3 to 9-8
- Indexed files
 - see indexed organization files
- INDEX function, 10-25, 10-26, 16-6
- Indexed I/O statements
 - READ statements, 11-37 to 11-39
 - WRITE statements, 11-47 to 11-49
- Indexed organization files
 - access keys
 - specifier in OPEN statement, 13-12, 13-13
 - creating, 15-2 to 15-3
 - deleting records from, 15-7
 - DELETE statement, 13-28 to 13-29
 - exception conditions, 15-7 to 15-8
 - freeing locked records
 - UNLOCK statement, 13-30
 - general description, 11-11
 - reading operations, 15-5 to 15-6
 - record pointers
 - next and last, 15-7
 - rewrite operations, 11-51
 - sequential access, 15-1
 - updating records in, 15-6
 - writing operations, 15-3 to 15-5
 - see also keyed access
- Indexed sequential access method (ISAM), 15-1
- INITIALSIZE
 - OPEN statement keyword, 13-4, 13-11
- Input/output
 - see I/O
- INQUIRE command (DCL), 1-35
- INQUIRE statement
 - general description, 13-19 to 13-26
- Integer
 - constants, 6-5
 - in COMPLEX*8 constants, 6-9
 - in COMPLEX*16 constants, 6-10
 - in REAL*4 constants, 6-6, 6-7
 - in REAL*8 constants, 6-7, 6-8
 - in REAL*16 constants, 6-8, 6-9
 - octal notation, A-1, A-6
 - setting default length, 3-11
- Integer, (Cont.)
 - data type
 - default type of undeclared symbolic names, 8-17
 - definition, 6-1
 - representation in memory, C-1, C-2
 - storage requirements, 6-2
 - see also arrays, constants, data types, variables
- INTEGER
 - see integer
- INTEGER*n
 - see integer
- Integer editing (I,O,Z), 12-6 to 12-9
- Interactive program execution
 - continuing, 4-7
 - interrupting, 4-7, 4-8
- Internal files, 11-11
- Internal file specifier
 - control list parameter
 - in I/O statements, 11-17
- Internal I/O statements, 11-11
 - ENCODE and DECODE statements, A-1 to A-3
 - READ statements, 11-39 to 11-41
 - WRITE statements, 11-49 to 11-50
- Interprocess communication, 11-3
- Interrupting program execution
 - <CTRL/c> and <CTRL/y>, 4-7 to 4-8
- Intrinsic functions, system-supplied
 - algorithms used in, D-30
 - complete list of, D-30 to D-38
 - character comparison functions, 10-24 to 10-26
 - description of types, 10-1
 - external procedures of same name, 8-17
 - lexical comparison functions, 10-26 to 10-27
 - references, generic, 10-20 to 10-21, 10-22 to 10-24
 - references, specific, 10-19 to 10-20, 10-22 to 10-24
 - use of names as arguments, 8-18
- Intrinsic function references
 - specific and generic, 10-19 to 10-24
- INTRINSIC statement
 - general description, 8-18 to 8-19
- I/O
 - character I/O, 16-7 to 16-10

I/O, (Cont.)
controlling I/O in command procs, 1-34 to 1-35
internal I/O, 11-11
I/O records, 11-12 to 11-14

I/O, internal
see internal I/O

I/O records
general description, 11-3
see also records (I/O)

IOSTAT
specifier in I/O statements, 11-21, 18-1, 18-6 to 18-7
BACKSPACE statement keyword, 13-27
CLOSE statement keyword, 13-18
DELETE statement keyword, 13-28, 13-29
ENDFILE statement keyword, 13-27, 13-28
INQUIRE statement specifier, 13-19, 13-22
OPEN statement keyword, 13-4, 13-11
REWIND statement keyword, 13-26
UNLOCK statement keyword, 13-30

I/O statements
forms of, 11-3 to 11-4
list of, 11-16
OPEN statement interdependencies
file organization, 11-10
file specification, 11-8 to 11-9
logical unit specifier, 11-17
specifiers
see I/O statement components
types of access modes, 11-2
use of unsubscripted arrays with, 6-21
see also, I/O statement components,
ACCEPT, FORMAT, OPEN, PRINT,

READ,
REWRITE, TYPE, WRITE

I/O statement components
control list parameters, 11-16,
rules for specifying, 11-22 to 11-23
format specifier, 11-17 to 11-18
internal file specifier, 11-17
I/O status specifier, 11-21
key-field-value specifier, 11-19 to 11-20
key-of-reference specifier, 11-20 to 11-21
logical unit specifier, 11-17
namelist specifier, 11-18
record specifier, 11-19
transfer-of-control specifier, 11-21 to 11-22

I/O list parameter, 11-23
implied-DO lists, 11-24 to 11-26
interaction with format controls, 12-28 to 12-29
simple list elements, 11-23 to 11-24

I/O status specifier
control list parameter
in I/O statements, 11-21

I/O units
see logical I/O units

ISAM, 15-1

Iterative I/O
implied-DO list, 11-24 to 11-26
iterative count controls
indexed DO statement, 9-3 to 9-6

Iterative processing controls
see DO statements

/I4 qualifier, 3-11

J

JOB logical name table, 1-18 to 1-19

.JOU files
see journal files

Journal files (EDT)
recovery operations, 2-5 to 2-6

K

'KEEP'
file disposition, 13-18

KEY
INQUIRE statement specifier, 13-19, 13-22 to 13-22
key-field-value specifier
in I/O statements, 11-19 to 11-20
OPEN statement keyword, 13-4, 13-12 to 13-13

KEYx specifier (KEY, KEYEQ, KEYGE, KEYGT)
see key-field-value specifier

KEYID specifier
see key-of-reference specifier

Keys, access
specified in OPEN statement, 13-12 to 13-13

Keys, primary and alternate
see key fields

'KEYED'
 OPEN statement keyword value, 13-3, 13-6

Keyed access mode, 11-15
 general discussion, 15-1 to 15-8
 see also indexed organization files

Key fields
 primary and alternate
 definition, 15-2
 discussion of use, 15-2 to 15-5

Key-field-value specifier
 control list parameter (KEY, KEYEQ, KEYGE, KEYGT)
 in I/O statements, 11-19 to 11-20

Key-of-reference specifier
 control list parameter (KEYID)
 in I/O statements, 11-20 to 11-21

Keypad diagrams
 VT52, 2-8
 VT100, 2-8
 VT100, Debug command layout, 17-7
 VT200, 2-9

L

L edit descriptor, 12-18

Labels
 see statement labels

.LIST
 see relational operators

LEN function, 10-24, 10-26, 16-6

Length
 default for INTEGER and LOGICAL (/I4), 3-11
 of records
 see fixed-length records, variable-length records, segmented records, stream records
 of source lines (/EXTENDSOURCE), 3-9
 affect on sequence number fields, 5-11
 specifier in data type declarations, 6-2

Lexical comparison library functions
 LLT, LLE, LGT, LGE, 10-26 to 10-27, 16-7

LGE function, 10-26 to 10-27, 16-7

LGT function, 10-26 to 10-27, 16-7

Libraries
 see text file libraries, intrinsic functions

LIBRARY command (DCL), 3-15 to 3-17

/LIBRARY qualifier
 FORTRAN command, 3-11
 LINK command, 4-3, 4-5

Library search order
 during compilation, 3-18

%LINE prefix
 SET BREAK command (Debug), 17-29

Lines, FORTRAN source code
 all blank, 6-10
 entry methods
 fixed format, 5-7 to 5-8
 tab format, 5-9 to 5-10
 format requirements
 continuation indicator field, 5-11
 sequence number field, 5-11
 statement field, 5-11
 statement label field, 5-10
 space characters embedded in, 5-7

Line mode editing (EDT), 2-1, 2-12 to 2-23
 command summary, 2-12 to 2-14
 deleting text, 2-19
 displaying text, 2-17 to 2-18
 getting text from other files, 2-21 to 2-22
 inserting text, 2-18 to 2-19
 line positioning, 2-18
 moving text, 2-20
 replacing text, 2-19
 range specification, 2-14 to 2-17
 string substitution, 2-20 to 2-21
 writing text to other files, 2-21 to 2-22

Line speed
 see SET TERMINAL command

LINK command
 format, 4-2
 options, 4-2 to 4-5

Linker
 compiler input for, 3-1 to 3-2
 messages, 4-6

LIS
 file type, 3-11

Lists, implied-DO
 use in DATA statements, 8-5

List-directed formatting
 I/O statement specifier, 11-17 to 11-18

List-directed I/O statements
 ACCEPT statement, 11-52
 general description, 11-3
 READ statements
 internal READ, 11-39, 11-40

- List-directed I/O statements, (Cont.)
 - sequential READ, 11-26, 11-28 to 11-30
 - WRITE statements
 - internal WRITE, 11-49, 11-50
 - sequential WRITE, 11-41, 11-43 to 11-44
- List elements, simple
 - I/O list parameter
 - in I/O statements, 11-23 to 11-24
- Listing file
 - see output listing
- LLE function, 10-26 to 10-27, 16-7
- LLT function, 10-26 to 10-27, 16-7
- %LOC built-in function, 10-8
- Local symbols
 - assignment operations 1-8, 1-29
 - controlling availability of, 3-8
 - local symbol definitions
 - effect of /DEBUG, 3-8
- Locked records
 - freeing locked records
 - UNLOCK statement, 13-30
- Logical
 - constants
 - .TRUE. and .FALSE., 6-13
 - representation in memory, C-1
 - setting default length, 3-11
 - storage requirement, 6-2
 - data type
 - definition, 6-1
 - relationship to BYTE data type, 8-7
 - see also arrays, constants, data types, logical values, variables
- LOGICAL*n
 - see logical
- Logical assignment statement, 7-3 to 7-4
- Logical editing (L), 12-18
- Logical elements
 - see logical expressions
- Logical expressions, 6-39 to 6-40
 - compile-time logical expressions
 - use in PARAMETER statements, 8-21, 8-22
- Logical IF statement, 9-13 to 9-14
- Logical I/O units
 - CLOSE statement options, 13-18
 - connection method
 - explicitly by means of OPEN, 13-17
 - implicitly by system default, 11-7 to 11-8

- Logical I/O units, (Cont.)
 - default FORTRAN logical unit numbers, 11-7 to 11-8
 - defining logical unit numbers
 - DEFINE FILE statement, A-1, A-3 to A-4
 - general discussion, 11-2
 - inquiring about properties
 - INQUIRE statement, 13-19 to 13-26
 - OPEN statement options, 11-8, 13-2
 - system unit numbers and names, 11-5 to 11-6
 - see also system logical names
- Logical names
 - see FORTRAN logical names, system logical names, logical I/O units
- Logical name tables, 1-18 to 1-19
- Logical operators, 6-39, 6-40
- Logical scalar memory reference
 - see scalar memory reference
- Logical units
 - see logical I/O units, UNIT
- Logical unit specifier
 - control list parameter
 - in I/O statements, 11-17
- Logical values
 - representation in memory, C-8 to C-9
- Log-in command file, 1-38
- Log-in procedure, 1-4
 - accessing other nodes, 1-6 to 1-7
- Log-out procedure, 1-4
- Loops, DO
 - DO statements, 9-3 to 9-10
- Lowercase characters
 - flagging in output
 - SOURCEFORM parameter (/STANDARD), 3-14
 - in character and Hollerith constants, 5-7
 - supported by VAX FORTRAN, 5-6
 - see relational operators

M

- Machine code listing
 - general description, 3-29 to 3-31
 - representation in MACRO, 3-12
- /MACHINE__CODE qualifier, 3-12
- MACRO code
 - use to represent object code, 3-12

MACRO code, (Cont.)
 unsupported codes generated by
 FORTRAN, 3-12

Main program
 see program unit

Map declaration
 general description, 8-31 to 8-33
 use to establish variables, 6-16

MAP parameter
 /SHOW qualifier, 3-13
 LINK command, 4-2, 4-4 to 4-5

MAP statement, 8-31, 8-32

Mathematical functions, intrinsic
 see subprograms

MAXREC
 OPEN statement keyword, 13-4, 13-13

Memory diagrams
 structured records, 6-25, 6-26

Messages
 compiler-fatal diagnostic messages, E-26
 to E-27
 CDD and CDDL error messages, E-41
 CRX error messages, E-41 to E-43
 linker messages, 4-6
 run-time messages, E-28 to E-40
 sending to terminal
 see PAUSE statement
 source program diagnostic messages, E-1
 to E-26
 warning and informational
 /STANDARD qualifier, 3-14
 /WARNING qualifier, 3-15

Minus operator (-), 6-33 to 6-34, 6-40,
 17-34

MOUNT command (DCL)
 logical name assignment, 1-20 to 1-21

MTH\$
 prefix for condition symbols
 for FORTRAN run-time errors, 18-5

Multiplication operator (*), 6-33 to 6-34

MVBITS subroutine, D-42 to D-43

N

NAME
 INQUIRE statement specifier, 13-19, 13-23
 OPEN statement keyword, 13-4, 13-13

NAMED
 INQUIRE statement specifier, 13-19, 13-23

Names
 see symbolic names, entry names

Named common blocks
 establishing order of contents, 8-3 to 8-4
 initializing values in, 8-2 to 8-3

Namelist specifier
 control list parameter
 in I/O statements, 11-18

NAMELIST statement
 general description, 8-20
 use of unsubscripted arrays with, 6-21

Namelist-directed I/O statements
 ACCEPT statement, 11-52
 general description, 11-3
 sequential READ statement, 11-27,
 11-30 to 11-34
 sequential WRITE statement, 11-41,
 11-44 to 11-45
 see also NAMELIST statement

.NE.
 see relational operators

.NEQV.
 see logical operators

Nested block IF constructs, 9-19 to 9-20

Nested DO loops, 9-6 to 9-8

Nested structured declarations
 see substructure declarations

Networks
 accessing other nodes, 1-6 to 1-7
 SET HOST, 1-7
 SHOW NETWORK, 1-6

NEXTREC
 INQUIRE statement specifier, 13-19,
 13-24

NML specifier
 in I/O statements, 11-18

Nodes
 file spec default, 1-16, 11-5
 see also networks

Nonexecutable statements
 definition, 5-2

NOSPANBLOCKS
 OPEN statement keyword, 13-4, 13-13

.NOT.
 see logical operators

NUMBER
 INQUIRE statement specifier, 13-19,
 13-24

Numbers, sequence, 5-11

Number generator, random, D-42

Numeric scalar memory reference
 see scalar memory reference
Numeric storage unit, 6-2
Numeric type declarations
 general description, 8-7 to 8-8

O

O field descriptor, 12-8
Object files
 naming (/OBJECT), 3-12
 see also compilation options
/OBJECT qualifier, 3-3, 3-4, 3-12
Octal constants, 6-11 to 6-12
 data typing of, 6-12
Octal notation ("")
 for integer constants, A-1, A-6
Octal values
 I/O transfers
 by O field descriptor, 12-8
ON...THEN command (DCL), 1-36, 1-37
OPEN statement
 general description, 13-1 to 13-18
 I/O statement interdependencies, 11-2,
 11-8 to 11-9, 11-17
 file organization, 11-10
 file specification, 11-8, 11-9
 logical unit specifier, 11-17
OPENED
 INQUIRE statement specifier, 13-19,
 13-24
Operators
 in DCL expressions, 1-30
 expression operators
 summary of, D-1 to D-2
 precedence in logical expressions, 6-40
 see also arithmetic operators, relational
 operators
Optimization
 affect on debugging, 3-8
 effect of VOLATILE statement, 8-34
 /OPTIMIZE qualifier, 3-12
/OPTIMIZE qualifier, 3-12
/OPTIONS qualifier (LINK), 4-3
OPTIONS statement, 3-26
.OR.
 see logical operators
Order
 required statement order, 5-3

ORGANIZATION

 INQUIRE statement specifier, 13-19,
 13-24 to 13-25
 OPEN statement keyword, 13-4, 13-13
OUT
 Debug screen display, 17-38, 17-39
Output files
 see listing files
Output listing, 3-28 to 3-35
 qualifiers affecting output
 see compilation options
Output options
 see compilation options
Overflow
 see arithmetic overflow
OWNER file access, 1-25

P

P edit descriptor, 12-16 to 12-17
Parameters
 passing to command procs, 1-32 to 1-33
PARAMETER statement, 8-21 to 8-22
 alternate version of, A-1, A-5 to A-6
Passed-length character arguments, 6-17,
 10-5 to 10-6, 16-4 to 16-5
Passed-length format, *(*)
 for dummy arguments or character
 functions, 6-3
Passwords
 changing, 1-5
Pathname prefixes (Debug)
 making symbolic references unique, 17-30
Pathnames (CDD), 3-27
PAUSE statement, 9-20 to 9-21
Period (.)
 current location indicator (Debug), 17-28
 operand contents specifier (Debug), 17-34
Plus operator (+), 6-33 to 6-34, 6-40, 17-34
Positional editing (X,T,TL,TR), 12-22 to
 12-23
Precedence, operator
 within arithmetic expressions, 6-33
Prefixes (Debug)
 making symbolic references unique, 17-29
 to 17-30
PREPROCESSOR parameter
 of /SHOW qualifier, 3-13

Pretested indefinite DO statement
 DO WHILE, 9-9
 Primary key fields
 see key fields
 'PRINT'
 file disposition, 13-18
 PRINT command (DCL), 1-26 to 1-27
 PRINT statement, 11-53
 PROCESS logical name table, 1-18
 Programs, main
 see program unit
 Program debugging
 see debugger
 Program execution, 4-6 to 4-11
 controlling during debug session, 17-17
 to 17-23
 interrupting
 <CTRL/C> and <CTRL/Y>, 4-7 to 4-8
 temporarily suspending (PAUSE), 9-20 to
 9-21
 terminating
 EXIT, D-40
 STOP, 9-23
 Program section (PSECT), 3-32
 PROGRAM statement
 general description, 8-23
 Program unit
 assigning symbolic name
 to main program unit, 8-23
 block data program unit, 8-2
 definition of, 5-2
 Protection
 codes (RWED), 1-25
 handling file protections, 1-25 to 1-27
 for source files when creating (EDT), 2-5
 PSECT, 3-32
 PURGE command (DCL), 1-23 to 1-24

Q

Q edit descriptor, 12-24
 Qualifiers
 rules for specifying, 1-3
 Question mark
 namelist prompt, 11-32
 Quotation marks (")
 octal notation for integer constants, A-1,
 A-6

R

Radix
 specifying during debug session,
 17-36 to 17-37
 Radix-50
 constants and character sets, B-3
 to B-4
 RAN function, D-42
 Random number generator
 RAN function, D-42
 Range specifications (EDT), 2-14 to
 2-17
 READ command (DCL), 1-35
 READ statements
 direct access READ, 11-35
 formatted, 11-36
 unformatted, 11-37
 indexed READ, 11-37
 formatted, 11-38
 unformatted, 11-38 to 11-39
 internal READ, 11-39
 formatted, 11-40
 list-directed, 11-41 to 11-42
 sequential READ, 11-26
 formatted, 11-27
 list-directed, 11-28 to 11-30
 namelist-directed, 11-30 to 11-34
 unformatted, 11-34 to 11-35
 relationship to DECODE statement, A-1,
 A-2
 READONLY
 OPEN statement keyword, 13-4, 13-14
 REAL
 see REAL*4
 REAL*4
 constants, 6-6 to 6-7
 data type
 default for undeclared symbolic names,
 8-17
 definition, 6-1
 representation in memory, C-2 to C-3
 storage requirements, 6-2
 see also arrays, constants, data types,
 variables
 REAL*8
 constants, 6-7 to 6-8
 G_floating vs. D_floating, 3-10, 6-3
 data type
 definition, 6-1

REAL*8, (Cont.)

representation in memory (G__ and D__floating), C-2, C-3 to C-4

storage requirement, 6-2

see also arrays, constants, data types, variables

REAL*16

constants, 6-8 to 6-9

data type

definition, 6-1

representation in memory, C-2, C-4 to C-5

storage requirements, 6-2

see also arrays, constants, data types, variables

Real editing (F,E,D,G), 12-10 to 12-16

complex data editing, 12-15 to 12-16

relationship to DECODE statement, A-1, A-2

REC

DELETE statement keyword, 13-28, 13-29
specifier in I/O statements, 11-19

RECL

INQUIRE statement specifier, 13-19, 13-25

OPEN statement keyword, 13-4, 13-14
to 13-15

Records

I/O records

general description, 11-3

deleting records from a file (DELETE),
13-28 to 13-29, 15-7

freeing locked records (UNLOCK), 13-30

in internal files, 11-11

ISAM record pointers (current and
next), 15-7

record formats (fixed-length, segmented,
stream, variable-length), 11-12 to
11-14

record operations on indexed files, 15-3
to 15-7

RECORDTYPE keyword (INQUIRE
statement), 13-19, 13-25

RECORDTYPE keyword (OPEN
statement), 13-15 to 13-16

sizes (OPEN statement keywords), 13-14
to 13-15

structured data items

arrangement in memory, 6-24 to 6-26

contrasted with arrays, 6-23

Records, (Cont.)

overview, 6-23, 14-3 to 14-4

Record access modes

direct, 11-15

keyed, 11-15

sequential, 11-14 to 11-15

Record and field references

aggregate field reference

definition, 6-27 to 6-28

use in unformatted I/O, 6-29

scalar field reference

definition, 6-27 to 6-28

Record specifier

control list parameter

in I/O statements, 11-19

RECORDSIZE

OPEN statement keyword, 13-4, 13-15

RECORD statement

general description, 8-23 to 8-24, 14-3

RECORDTYPE

INQUIRE statement specifier, 13-19, 13-25

OPEN statement keyword, 13-4, 13-15 to
13-16

Recovery

source files (EDT), 2-5

%REF built-in function, 10-7 to 10-8

use with Debug, 17-19

References, function

see function references

References, generic or specific

see function references

REG

Debug screen display, 17-38, 17-39

Relational expressions, 6-38 to 6-39

Relational operators, 6-38, D-1

avoiding use as field names

in structure declarations, 6-29

Relative files

see relative organization files

Relative organization files

deleting records from

DELETE statement, 13-28 to 13-29

defining size and structure

DEFINE FILE statement, A-1,
A-3 to A-4

freeing locked records in

UNLOCK statement, 13-30

general description, 11-10 to 11-11

see also direct access

RENAME command (DCL), 1-22, 1-25

Repeat counts
in FORMAT statements, 12-3

Return argument, alternate, 10-6 to 10-7

RETURN statement
general description, 9-21 to 9-22
use with CALL statements, 9-21, 9-22
use with FUNCTION statement, 10-11
use with SUBROUTINE statement, 10-14, 10-16
when to avoid, 4-8

Returning completion status values
to a command proc, 4-8

REWIND statement
general description, 13-26
see also BACKSPACE statement

REWRITE statements, 11-50 to 11-52

RTL
see Run-Time Library

RUN command (DCL), 4-6 to 4-7
using to invoke the debugger, 17-3

Running FORTRAN programs, 4-6 to 4-11

Run-time formats, 12-27 to 12-28

Run-Time Library
error processing performed by, 18-1 to 18-7

RWED
file protection codes, 1-25

S

S edit descriptor, 12-6

'SAVE'
file disposition, 13-18

SAVE statement
general description, 8-24 to 8-25
use of unsubscripted arrays with, 6-21

Scalar field reference
see record and field references

Scalar memory reference
definition and examples, 6-31 to 6-32

Scalar reference
definition and examples, 6-31 to 6-32

Scale factor editing (P), 12-16 to 12-17

Scope (Debug)
making symbolic references unique, 17-29
to 17-30, 17-31

Screen displays (Debug), 17-39 to 17-43

SCROLL command (Debug), 17-41

SEARCH command (DCL), 1-26

SEARCH command (Debug), 17-23, 17-24

Search lists, 1-21

SECNDS function subprogram, D-41

Segmented records
format, 11-13

RECORDTYPE keyword (OPEN statement), 13-4, 13-15 to 13-16

SELECT/SCROLL command (Debug), 17-41
key used in keypad mode, 17-43

Separators
external field separators, 12-27
format specification separators, 12-26

Sequence numbers, 5-8, 5-11

'SEQUENTIAL'
OPEN statement keyword value, 13-3, 13-6

SEQUENTIAL
INQUIRE statement specifier, 13-19, 13-25 to 13-26

Sequential access mode, 11-14 to 11-15
for indexed organization files, 15-1

Sequential files
see sequential organization files

Sequential I/O statements
READ statements, 11-26 to 11-35
WRITE statements, 11-41 to 11-46

Sequential organization files
repositioning
BACKSPACE statement, 13-27
REWIND statement 13-26
general description, 11-10
writing end-of-file records
ENDFILE statement, 13-27 to 13-28

SET commands (DCL)
SET DEFAULT, 1-15
SET HOST, 1-7
SET [NO]ON, 1-37, 1-38
SET PASSWORD, 1-5
SET PROTECTION, 1-25
SET TERMINAL, 1-5 to 1-6
SET VERIFY, 1-36

SET commands (Debug)
SET BREAK, 17-20 to 17-22
SET DISPLAY, 17-41 to 17-42
SET LOG, 17-10
SET MODE radix, 17-36 to 17-37
SET MODE SCREEN, 17-23, 17-24, 17-39

SET commands (Debug), (Cont.)

- SET MODULE, 17-26 to 17-27
- SET OUTPUT LOG, 17-10
- SET SCOPE, 17-31
- SET TRACE, 17-20, 17-23
- SET TYPE/OVERRIDE, 17-35
- SET WATCH, 17-20 to 17-22
- SET WINDOW, 17-40

/SHAREABLE qualifier (LINK), 4-2, 4-4

SHARED

- OPEN statement keyword, 13-4, 13-16

SHOW commands (DCL)

- SHOW DEFAULT, 1-15
 - SHOW NETWORK, 1-6
 - SHOW QUOTA, 1-15
- ## SHOW commands (Debug)
- SHOW BREAK, 17-22
 - SHOW CALLS, 17-19 to 17-20
 - SHOW KEY, 17-7, 17-8
 - SHOW MODULE, 17-26, 17-27
 - SHOW STEP, 17-18
 - SHOW SYMBOL, 17-31 to 17-32
 - SHOW TRACE, 17-22
 - SHOW TYPE, 17-36
 - SHOW WATCH, 17-23
 - SHOW WINDOW, 17-40

/SHOW qualifier, 3-13 to 3-14

Sign control editing, 12-6

Simple list elements

- I/O list parameter
 - in I/O statements, 11-23 to 11-24

SINGLE parameter (/SHOW), 3-13

Slash (/)

- division operator, 6-33 to 6-34, 6-40, 17-34
- record terminators
 - in FORMAT statements, 12-1

Source code listing

- general description, 3-28 to 3-29
- qualifiers affecting output
 - /LIST, 3-11
 - /SHOW, 3-13

Source code

- allowable characters, 5-7 to 5-7
- coding restrictions/limits
 - summary of, E-27 to E-28
- comments, 3-9, 5-5
- debugging statements in, 5-10

Source code, (Cont.)

- field descriptions, 5-7 to 5-11
- format requirements
 - fixed-format lines, 5-7 to 5-8
 - tab-format lines, 5-9 to 5-10
- see also source programs

Source files

- creating (EDT), 2-6
- input to compiler, 3-2 to 3-3
- protecting (EDT), 2-5 to 2-6
- recovering (EDT), 2-5 to 2-6
- see also files

SOURCE__FORM parameter (/STANDARD), 3-14

Source programs

- compile options
 - continuation line limits, 3-7
 - D in column 1, 3-9, 5-10
 - maximum line length, 3-9
- program unit
 - definition, 5-2
- statement order, 5-2 to 5-3
- symbolic names
 - rules, conventions, and use, 5-4 to 5-5
- see also source code

Source program diagnostic messages, E-1 to E-26

SP

- edit descriptor, 12-6

Space characters

- effect of FORMAT descriptors, 12-3, 12-5
- in character and Hollerith constants, 5-7
- in source code in general, 5-7
- in statement label fields, 5-10

Special characters

- FORTRAN supported, 5-6
- Specification statements, 8-1 to 8-34

SRC

- Debug screen display, 17-38, 17-39

SS

- edit descriptor, 12-6

SS\$

- prefix for condition symbols for FORTRAN run-time errors, 18-4 to 18-5

/STANDARD qualifier, 3-14

Standards

- see ANSI standards, FORTRAN-66, FORTRAN-77

Startup command files (EDT), 2-25 to 2-26

Statements, FORTRAN
 see FORTRAN statements

Statement function subprograms
 see subprograms

Statement labels
 rules governing use, 5-2, 5-8, 5-10, 7-6

Statement label references
 use in FORMAT and GOTO statements,
 7-6 to 7-7

STATUS
 CLOSE statement keyword, 13-18
 OPEN statement keyword, 13-4, 13-16 to
 13-17

Status values, completion
 returning to a command proc, 4-8

STEP commands (Debug), 17-18 to 17-19
 STEP/SOURCE, 17-23, 17-24

STOP statement
 affect on program execution, 4-7
 general description, 9-23
 when not to use, 4-8

Storage allocation, file
 OPEN statement keywords
 EXTENDSIZE, 13-3, 13-10
 INITIALSIZE, 13-4, 13-11

Storage map listing
 general description, 3-32 to 3-35
 qualifiers affecting output
 /SHOW, 3-13

Storage units
 character, 6-2
 numeric, 6-2

Stream records
 format, 11-13 to 11-14
 RECORDTYPE keyword (OPEN
 statement), 13-4, 13-15 to 13-16

Structures
 general description, 14-2
 see also records (structured data items),
 structure declaration blocks,
 substructure declarations

Structure declaration blocks
 CDD data definitions, relationship to,
 3-20, 3-21
 data type declaration rules, 8-28
 field declarations within, 8-25,
 8-26, 8-27 to 8-33
 general description, 8-25 to 8-33

Structure declaration blocks, (Cont.)
 overview, 6-23 to 6-24, 14-2
 use of %FILL, 6-24, 8-28
 see also substructure declarations

STRUCTURE statement
 general description, 8-25, 8-26, 14-2

Subdirectories, 1-14 to 1-17
 creating, 1-15
 deleting, 1-17

'SUBMIT'
 file disposition, 13-18

SUBMIT command (DCL), 1-37 to 1-38
 /PARAMETERS qualifier, 1-33

Subprograms
 definition of, 5-2
 bit functions
 general discussion about, D-43 to D-45

CALL statement, 9-2

CHARACTER FUNCTION statement,
 10-12

ENTRY statement, 10-16 to 10-19

FUNCTION statement, 10-11 to 10-14

functions, built-in
 argument list functions (%VAL, %REF,
 %DESCR), 10-7 to 10-8
 %LOC function, 10-8

function references, 10-13 to 10-14

RETURN statement, 9-21 to 9-23

system-supplied FORTRAN intrinsic
 functions
 algorithms used in, D-30
 complete list of, D-30 to D-38
 character comparison functions, 10-24 to
 10-26
 description of types, 10-1
 duplicating external procedure names,
 8-17
 lexical comparison functions, 10-26 to
 10-27
 references, generic, 10-20 to
 10-21, 10-22 to 10-24
 references, specific, 10-19 to 10-20,
 10-22 to 10-24
 use of names as arguments, 8-18

system-supplied subroutines and functions
 list and descriptions of, D-38 to D-43

user-written functions
 function subprograms, 10-11 to 10-14
 ENTRY statements in, 10-17 to 10-18

RETURN statement, (Cont.)

- statement functions, 10-9 to 10-11
- subroutine subprograms, 10-14 to 10-16
- ENTRY statements in, 10-18 to 10-19

Subprogram arguments (actual and dummy arguments)

- general description, 10-2 to 10-8
- overview, 10-2
- adjustable arrays, 10-3 to 10-4
- alternate return arguments, 10-6 to 10-7
- assumed-size arrays, 10-4 to 10-5
- character arrays, 10-5 to 10-6
- defaults for arguments passing, 10-7 to 10-8
- Hollerith and character constants, 10-6
- passed-length character arguments, 10-5 to 10-6, 16-4 to 16-5
- associating variables with, 6-16
- bit function arguments, D-43 to D-45
- use of aggregate field references, 6-29
- use of built-in functions
 - argument list functions (%VAL, %REF, %DESCR), 10-7 to 10-8
 - %LOC function, 10-8
- use of external procedure names, 8-16 to 8-17
- use of intrinsic function names, 8-18

Subroutine arguments

- see subprogram arguments

SUBROUTINE statement, 10-14 to 10-15

- see also subprograms

Subscripts

- see arrays

Substrings, character

- see character substrings

Substring equivalence, 8-13 to 8-16

Substring references

- checking boundaries
- FORTTRAN command option, 3-6

Substructure declarations

- general description, 6-24, 8-25, 8-31

Summary listing, compilation, 3-35

Symbolic names

- assigning to constants
 - with PARAMETER statement, 8-21 to 8-22
- assigning to main program unit, 8-23
- default data types assigned, 8-17
- reference information in listing file, 3-7

Symbolic names, (Cont.)

- referencing during debug session, 17-25 to 17-27
- rules, conventions, and use, 5-4 to 5-5
- use as arguments to subprograms
 - external procedure names, 8-16 to 8-17
 - intrinsic function names, 8-18 to 8-19
- use with arrays, 6-18
- use with variables, 6-15

Symbols

- assignment operations 1-8, 1-29
- controlling availability of, 3-8
- local symbol definitions
 - effect of /DEBUG, 3-8

Symbols (DCL)

- use in commands procs, 1-28 to 1-32

Symbol map

- MAP parameter (/SHOW), 3-13
- SYNTAX parameter (/STANDARD), 3-14
- SYS\$COMMAND, 1-19, 1-34
- SYS\$DISK, 1-19
- SYS\$ERROR, 1-19
- SYS\$INPUT, 1-19, 1-34
- SYS\$LIBRARY, 3-19
- SYS\$LOGIN, 1-19
- SYS\$NODE, 1-19
- SYS\$OUTPUT, 1-19, 1-34
- SYS\$SCRATCH, 1-19
- /SYSLIB qualifier (LINK), 4-3
- /SYSSHR qualifier (LINK), 4-3

System access, 1-3 to 1-6

- see also log-in procedure, log-out procedure

System definition modules, 3-19

SYSTEM file access, 1-25

System logical names

- assigning with MOUNT, 1-20
- defaults, 1-19
- defining, 1-18
- deleting, 1-20
- displaying, 1-20
- search lists, 1-21
- using to identify devices, 1-11

System logical name table, 1-18 to 1-19

System time

- function subprogram for calculating SECNDS, D-41
- subroutine for calculating TIME, D-41 to D-42

T

T edit descriptor, 12-22 to 12-23
Tab character, 5-6
Tab formatting
 EDT set-up, 2-25 to 2-26
 flagging in output
 SOURCE_FORM (/STANDARD), 3-14
 general description, 5-9 to 5-10
Terminals
 assigning SYS\$COMMAND, SYS\$INPUT,
 or SYS\$OUTPUT, 1-34 to 1-35
 setting speed, 1-6
 setting type, 1-5
Text editor (EDT), 2-1 to 2-26
Text file libraries
 general discussion, 3-15 to 3-19
 accessing (INCLUDE), 3-24 to 3-25
 creating and modifying
 LIBRARY command (DCL), 3-15 to
 3-17
 defining defaults, 3-18 to 3-19
 /LIBRARY qualifier, 3-11
 search order, 3-18
 system-supplied default library
 FORSYSDEF.TLB, 3-19
TIME subroutine, D-41 to D-42
Time, system
 function subprogram for calculating
 SECNDS, D-41
 subroutine for calculating
 TIME, D-41 to D-42
TL
 edit descriptor, 12-22, 12-23
TLB
 file type, 3-15
TR
 edit descriptor, 12-22, 12-23
Traceback mechanism
 effect of /DEBUG
 FORTRAN command, 3-8
 LINK command, 4-2, 4-5
/TRACEBACK qualifier (LINK), 4-2, 4-5,
 4-9 to 4-11
Tracepoints
 setting during debug session, 17-20
 to 17-22
Transfer, control
 FORTRAN control statements, 9-1 to 9-23
 see also CALL, RETURN

Transfer, data
 see I/O
Transfer-of-control specifier
 control list parameter
 in I/O statements, 11-21 to 11-22
.TRUE.
 see logical constants
Type
 see file type, data type
TYPE
 OPEN statement keyword, 13-4, 13-17
TYPE command (DCL), 1-26 to 1-27
TYPE command (Debug), 17-23, 17-24
TYPE command (EDT), 2-17
Type declaration statement
 see data type declaration statement
TYPE statement, 11-53

U

Unary plus and minus operators (+ and -),
 6-33 to 6-34, 6-40
Unconditional GOTO statement, 9-10 to
 9-11
Undeclared symbolic names
 default data types, 8-17
UNFORMATTED
 INQUIRE statement specifier, 13-26
Unformatted I/O statements
 general description, 11-3
 relationship to ENCODE statement, A-1,
 A-2
 use of aggregate field references, 6-29
READ statements
 direct access, 11-35, 11-36
 indexed, 11-37, 11-38 to 11-39
 sequential, 11-27, 11-34 to 11-35
REWRITE statements, 11-50, 11-51 to
 11-52
WRITE statements
 direct access, 11-46, 11-47
 indexed, 11-47, 11-49
 sequential, 11-41, 11-45 to 11-46
Union declarations
 contrasted with EQUIVALENCE, 8-32
 definition, 8-25 to 8-26
 general description, 8-31 to 8-33
UNION statement, 8-31

Units, logical
 see logical I/O units
UNIT
 specifier in I/O statements, 11-17
 BACKSPACE statement keyword, 13-27
 CLOSE statement keyword, 13-18
 DELETE statement keyword, 13-28, 13-29
 ENDFILE statement keyword, 13-27,
 13-28
 INQUIRE statement keyword, 13-19
 OPEN statement keyword, 13-4, 13-17
 REWIND statement keyword, 13-26
 UNLOCK statement keyword, 13-30
UNLOCK statement, 13-30
Unnamed fields
 use of %FILL
 in structure declarations, 6-24
Unsubscripted arrays
 statements used in, 6-21
Uppercase characters
 in character and Hollerith constants, 5-7
 supported by VAX FORTRAN, 5-6
USER logical name table, 1-18
/USERLIBRARY qualifier (LINK), 4-3
USEROPEN
 OPEN statement keyword, 13-17 to 13-18
User-written subroutines
 see subprograms

V

%VAL built-in function, 10-7 to 10-8
 use with Debug, 17-19
Variables
 definition of, 6-15 to 6-16
 assigning values to
 with DATA statements, 8-4 to 8-6
 association of two or more
 with EQUIVALENCE statement, 6-16
 character substrings, 6-22 to 6-23
 data typing of
 by implication, 6-17
 by specification, 6-16 to 6-17
 defining in memory, 6-16
 establishing with subprogram references
 actual and dummy arguments, 6-16
 implied-DO variables
 in DATA statements, 8-5

Variables, (Cont.)
 initializing character variables, 8-4
 to 8-6, 16-4
 scalar reference, 6-31
 use in structure declarations, 6-24
Variable-length records
 format, 11-13
 RECORDTYPE keyword (OPEN
 statement),
 13-4, 13-15 to 13-16
VAX DEBUG
 see debugger
VAX FORTRAN
 extensions to ANSI standard, v, 5-1
VAX/VMS commands
 see DCL commands
Virtual memory locations
 defining symbolically (Debug), 17-31
 specifying program addresses (Debug),
 17-29
VIRTUAL statement, 8-10
VOLATILE statement
 general description, 8-34
VT52 keypad, 2-8
VT100 keypad, 2-8
 keypad layout for Debug commands, 17-7
VT200 keypad, 2-9

W

Warning messages
 see messages
/WARNINGS qualifier, 3-15
Watchpoints
 setting during a debug session,
 17-20 to 17-22
WHILE command (Debug), 17-25
Wildcards
 use in filespecs, 1-14
Windows (Debug)
 defining, 17-40
WORLD file access, 1-25
WRITE command (DCL), 1-35
WRITE statements
 direct access WRITE
 formatted, 11-46, 11-47
 unformatted, 11-46, 11-47
 indexed WRITE
 formatted, 11-47, 11-48

WRITE statements, (Cont.)
 unformatted, 11-47, 11-49
internal WRITE
 formatted, 11-49, 11-50
 list-directed, 11-49, 11-50
sequential WRITE
 formatted, 11-41, 11-42 to 11-43
 list-directed, 11-41, 11-43 to 11-44
 namelist-directed, 11-41, 11-44
 to 11-45
 unformatted, 11-41, 11-45 to 11-46

X

X edit descriptor, 12-22
.XOR.
 see logical operators

Z

Z field descriptor, 12-9

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

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
- Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

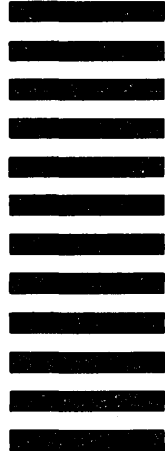
City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital



No Postage
Necessary
if Mailed in the
United States



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here