VAX FORTRAN
Language Reference Manual

Order Number: AA–D034E–TE

June 1988

This manual details the VAX FORTRAN programming language as implemented for VMS systems.

Revision/Update Information: This revised manual supersedes *Programming in VAX FORTRAN* (order number AA–D034D–TE).

Operating System and Version: VMS Version 5.0 or higher

Software Version: VAX FORTRAN Version 5.0

digital equipment corporation
maynard, massachusetts

First Printing, September 1984
Revision, June 1988

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, 1988 by Digital Equipment Corporation

All Rights Reserved.
Printed in U.S.A.

The postpaid Reader's Comments forms at the end of this document request the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem–10	PDP	VT
DECSYSTEM–20	PDT	
DECUS	RSTS	
DECwriter	RSX	

digital™

ZK-4671

Contents

PREFACE xvii

NEW AND EXPANDED LANGUAGE FEATURES xxi

CHAPTER 1 INTRODUCTION TO VAX FORTRAN 1–1

1.1 ELEMENTS OF FORTRAN SOURCE PROGRAMS 1–2
1.1.1 Program Units 1–2
1.1.2 Statements 1–2
1.1.3 Symbolic Names 1–5
1.1.4 Comments 1–7

1.2 CHARACTER SET 1–8

1.3 CODING RULES 1–9
1.3.1 Fixed-Format Lines 1–10
1.3.2 Tab-Format Lines 1–11
1.3.3 Statement Label Field 1–13
1.3.3.1 Comment Indicator • 1–13
1.3.3.2 Debugging Statement Indicator • 1–14
1.3.4 Continuation Indicator Field 1–14
1.3.5 Statement Field 1–14
1.3.6 Sequence Number Field 1–15

1.4 COMPILATION CONTROL STATEMENTS 1–15
1.4.1 DICTIONARY Statement 1–15
1.4.2 INCLUDE Statement 1–17
1.4.3 OPTIONS Statement 1–18

CHAPTER 2 DATA TYPES, DATA ITEMS, AND EXPRESSIONS 2–1

2.1 DATA TYPES 2–1
2.1.1 Storage Requirements 2–2
2.1.2 VAX Implementations of REAL*8 2–4

2.2 DATA ITEMS 2–4
2.2.1 Constants 2–5
2.2.1.1 Integer Constants • 2–5
2.2.1.2 Real Constants • 2–8
2.2.1.3 Complex Constants • 2–13
2.2.1.4 Octal and Hexadecimal Constants • 2–15
2.2.1.5 Logical Constants • 2–18
2.2.1.6 Character Constants • 2–18
2.2.1.7 Hollerith Constants • 2–19
2.2.2 Variables 2–22
2.2.2.1 Data Type by Specification • 2–23
2.2.2.2 Data Type by Implication • 2–24
2.2.3 Arrays 2–24
2.2.3.1 Array Declarators • 2–25
2.2.3.2 Array Subscripts • 2–27
2.2.3.3 Arrangement of Array Elements in Storage • 2–27
2.2.3.4 Data Type of an Array • 2–29
2.2.3.5 Array References without Subscripts • 2–29
2.2.3.6 Adjustable Arrays • 2–29
2.2.3.7 Assumed-Size Arrays • 2–30
2.2.4 Character Substrings 2–30
2.2.5 Records 2–31
2.2.5.1 Arrangement of Records in Storage • 2–33
2.2.5.2 References to Record Fields • 2–36
2.2.6 Terminology Used to Refer to Data Items 2–39

2.3 EXPRESSIONS 2–42
2.3.1 Arithmetic Expressions 2–42
2.3.1.1 Using Parentheses • 2–44
2.3.1.2 Data Type of an Arithmetic Expression • 2–45
2.3.2 Character Expressions 2–47
2.3.3 Relational Expressions 2–48
2.3.4 Logical Expressions 2–49

CHAPTER 3 ASSIGNMENT STATEMENTS 3–1

3.1 ARITHMETIC ASSIGNMENT STATEMENT 3–1

3.2 LOGICAL ASSIGNMENT STATEMENT 3–4

3.3 CHARACTER ASSIGNMENT STATEMENT 3–4

3.4 AGGREGATE ASSIGNMENT STATEMENT 3–6

3.5 ASSIGN STATEMENT 3–7

CHAPTER 4 SPECIFICATION STATEMENTS 4–1

4.1 BLOCK DATA STATEMENT 4–2

4.2 COMMON STATEMENT 4–3

4.3 DATA STATEMENT 4–5

4.4 DATA TYPE DECLARATION STATEMENTS 4–8
4.4.1 Numeric Type Declaration Statements 4–8
4.4.2 Character Type Declaration Statements 4–10

4.5 DIMENSION STATEMENT 4–12

4.6 EQUIVALENCE STATEMENT 4–13
4.6.1 Making Arrays Equivalent 4–14
4.6.2 Making Substrings Equivalent 4–17
4.6.3 EQUIVALENCE and COMMON Interaction 4–20

4.7 EXTERNAL STATEMENT 4–21

4.8 IMPLICIT STATEMENT 4–22

4.9 INTRINSIC STATEMENT 4–23

4.10 NAMELIST STATEMENT 4–24

4.11 PARAMETER STATEMENT 4–26

4.12 PROGRAM STATEMENT 4–28

4.13 RECORD STATEMENT 4–29

4.14 SAVE STATEMENT 4–30

4.15 STRUCTURE DECLARATION BLOCK 4–31
4.15.1 Structure Declaration 4–33
4.15.2 Substructure Declarations 4–38
4.15.3 Union Declarations 4–38

4.16 VOLATILE STATEMENT 4–41

CHAPTER 5 CONTROL STATEMENTS 5–1

5.1 CALL STATEMENT 5–2

5.2 CONTINUE STATEMENT 5–3

5.3 DO STATEMENTS 5–3
5.3.1 Indexed DO Statement 5–4
5.3.1.1 DO Iteration Control • 5–5
5.3.1.2 Nested DO Loops • 5–7
5.3.1.3 Control Transfers in DO Loops • 5–8
5.3.1.4 Extended Range • 5–9
5.3.2 DO WHILE Statement 5–9

5.4 END DO STATEMENT 5–11

5.5 END STATEMENT 5–12

5.6 GO TO STATEMENTS 5–12
5.6.1 Unconditional GO TO Statement 5–13
5.6.2 Computed GO TO Statement 5–13
5.6.3 Assigned GO TO Statement 5–14

5.7 IF STATEMENTS 5–15
5.7.1 Arithmetic IF Statement 5–16
5.7.2 Logical IF Statement 5–17
5.7.3 Block IF Statements 5–17
5.7.3.1 Statement Blocks • 5–21
5.7.3.2 Block IF Examples • 5–21
5.7.3.3 Nested Block IF Constructs • 5–23

5.8 PAUSE STATEMENT 5–24

5.9 RETURN STATEMENT 5–25

5.10 STOP STATEMENT 5–27

CHAPTER 6 SUBPROGRAMS — SUBROUTINES AND FUNCTIONS 6–1

6.1 SUBPROGRAM ARGUMENTS 6–2
6.1.1 Actual Argument and Dummy Argument Association 6–2
6.1.1.1 Adjustable Arrays • 6–3
6.1.1.2 Assumed-Size Arrays • 6–6
6.1.1.3 Passed-Length Character Arguments • 6–7
6.1.1.4 Character and Hollerith Constants as Actual Arguments • 6–8
6.1.1.5 Alternate Return Arguments • 6–9
6.1.2 Built-In Functions 6–9
6.1.2.1 Argument List Built-In Functions • 6–9
6.1.2.2 %LOC Built-In Function • 6–11

6.2 USER-WRITTEN SUBPROGRAMS 6–11
6.2.1 Statement Functions 6–12

6.2.2 **Function Subprograms** — 6–15
6.2.2.1 Logical and Numeric Functions • 6–15
6.2.2.2 Character Functions • 6–15
6.2.2.3 Function Reference • 6–16
6.2.3 **Subroutine Subprograms — SUBROUTINE Statement** — 6–18
6.2.4 **ENTRY Statement** — 6–21
6.2.4.1 ENTRY Statements in Function Subprograms • 6–22
6.2.4.2 ENTRY Statements in Subroutine Subprograms • 6–24

6.3 **FORTRAN INTRINSIC FUNCTIONS** — 6–25
6.3.1 **Intrinsic Function References** — 6–25
6.3.1.1 Generic References to Intrinsic Functions • 6–26
6.3.1.2 Using Intrinsic Function Names • 6–28
6.3.2 **Character and Lexical Comparison Library Functions** — 6–30
6.3.2.1 Character Functions • 6–30
6.3.2.2 Lexical Comparison Functions • 6–32

CHAPTER 7 I/O STATEMENTS — 7–1

7.1 **COMPONENTS OF I/O STATEMENTS** — 7–1
7.1.1 **Control List** — 7–2
7.1.1.1 Syntax Rules for Control-List Parameters • 7–3
7.1.1.2 Logical Unit Specifier • 7–3
7.1.1.3 Internal File Specifier • 7–4
7.1.1.4 Format Specifiers • 7–4
7.1.1.5 Namelist Specifier • 7–5
7.1.1.6 Record Specifier • 7–6
7.1.1.7 Key-Field-Value Specifier • 7–6
7.1.1.8 Key-of-Reference Specifier • 7–9
7.1.1.9 I/O Status Specifier • 7–9
7.1.1.10 Transfer-of-Control Specifiers • 7–10
7.1.2 **I/O List** — 7–11
7.1.2.1 Simple List Elements • 7–12
7.1.2.2 Implied-DO Lists in I/O Statements • 7–13

7.2 **READ STATEMENTS** — 7–15

7.2.1 **Sequential READ Statements** 7–15
7.2.1.1 Formatted Sequential READ Statement • 7–16
7.2.1.2 List-Directed Sequential READ Statement • 7–17
7.2.1.3 Namelist-Directed Sequential READ Statement • 7–20
7.2.1.4 Unformatted Sequential READ Statement • 7–25
7.2.2 **Direct Access READ Statements** 7–26
7.2.2.1 Formatted Direct Access READ Statement • 7–26
7.2.2.2 Unformatted Direct Access READ Statement • 7–27
7.2.3 **Indexed READ Statements** 7–28
7.2.3.1 Formatted Indexed READ Statement • 7–29
7.2.3.2 Unformatted Indexed READ Statement • 7–30
7.2.4 **Internal READ Statement** 7–31
7.2.4.1 Formatted Internal READ Statement • 7–31
7.2.4.2 List-Directed Internal READ Statement • 7–32

7.3 WRITE STATEMENTS 7–33
7.3.1 **Sequential WRITE Statements** 7–33
7.3.1.1 Formatted Sequential WRITE Statement • 7–34
7.3.1.2 List-Directed Sequential WRITE Statement • 7–35
7.3.1.3 Namelist-Directed Sequential WRITE Statement • 7–37
7.3.1.4 Unformatted Sequential WRITE Statement • 7–39
7.3.2 **Direct Access WRITE Statements** 7–39
7.3.2.1 Formatted Direct Access WRITE Statement • 7–40
7.3.2.2 Unformatted Direct Access WRITE Statement • 7–40
7.3.3 **Indexed WRITE Statements** 7–41
7.3.3.1 Formatted Indexed WRITE Statement • 7–42
7.3.3.2 Unformatted Indexed WRITE Statement • 7–43
7.3.4 **Internal WRITE Statement** 7–43
7.3.4.1 Formatted Internal WRITE Statement • 7–44
7.3.4.2 List-Directed Internal WRITE Statement • 7–44

7.4 REWRITE STATEMENT 7–45
7.4.1 **Formatted REWRITE Statement** 7–46
7.4.2 **Unformatted REWRITE Statement** 7–46

7.5 ACCEPT STATEMENT 7–47

7.6 TYPE AND PRINT STATEMENTS 7–48

CHAPTER 8 I/O FORMATTING 8–1

8.1 GENERAL RULES FOR WRITING FORMAT STATEMENTS 8–2
8.1.1 Input Rules for FORMAT Statements 8–3
8.1.2 Output Rules for FORMAT Statements 8–4

8.2 FORMAT STATEMENT SYNTAX 8–4

8.3 FIELD AND EDIT DESCRIPTORS 8–7
8.3.1 Repeat Counts and Group Repeat Counts 8–8
8.3.2 Variable Format Expressions 8–9
8.3.3 Blank Control Editing 8–10
8.3.3.1 BN Edit Descriptor • 8–10
8.3.3.2 BZ Edit Descriptor • 8–11
8.3.4 Sign Control Editing 8–11
8.3.4.1 SP Edit Descriptor • 8–11
8.3.4.2 SS Edit Descriptor • 8–11
8.3.4.3 S Edit Descriptor • 8–12
8.3.5 Integer Editing 8–12
8.3.5.1 I Field Descriptor • 8–12
8.3.5.2 O Field Descriptor • 8–14
8.3.5.3 Z Field Descriptor • 8–16
8.3.6 Real Editing 8–17
8.3.6.1 F Field Descriptor • 8–17
8.3.6.2 E Field Descriptor • 8–19
8.3.6.3 D Field Descriptor • 8–21
8.3.6.4 G Field Descriptor • 8–22
8.3.6.5 Complex Data Editing • 8–25
8.3.7 Scale Factor Editing—P Edit Descriptor 8–25
8.3.8 Logical Editing—L Edit Descriptor 8–28
8.3.9 Character Editing 8–29
8.3.9.1 A Field Descriptor • 8–29
8.3.9.2 H Field Descriptor • 8–32
8.3.9.3 Character Constants • 8–32
8.3.10 Default Field Descriptors 8–33
8.3.11 Positional Editing 8–34
8.3.11.1 X Edit Descriptor • 8–34
8.3.11.2 T Edit Descriptor • 8–35
8.3.11.3 TL Edit Descriptor • 8–36
8.3.11.4 TR Edit Descriptor • 8–36

8.3.12 Additional Editing Operations ____ 8–36
8.3.12.1 Q Edit Descriptor • 8–37
8.3.12.2 Dollar Sign Descriptor • 8–37
8.3.12.3 Colon Descriptor • 8–38

8.4 CARRIAGE CONTROL 8–38

8.5 FORMAT SPECIFICATION SEPARATORS 8–39

8.6 EXTERNAL FIELD SEPARATORS 8–40

8.7 RUN-TIME FORMAT 8–41

8.8 FORMAT CONTROL INTERACTION WITH I/O LISTS 8–42

CHAPTER 9 AUXILIARY I/O STATEMENTS 9–1

9.1 OPEN STATEMENT 9–2
9.1.1 ACCESS Keyword ____ 9–8
9.1.2 ASSOCIATEVARIABLE Keyword ____ 9–8
9.1.3 BLANK Keyword ____ 9–8
9.1.4 BLOCKSIZE Keyword ____ 9–9
9.1.5 BUFFERCOUNT Keyword ____ 9–9
9.1.6 CARRIAGECONTROL Keyword ____ 9–10
9.1.7 DEFAULTFILE Keyword ____ 9–10
9.1.8 DISPOSE Keyword ____ 9–11
9.1.9 ERR Keyword ____ 9–12
9.1.10 EXTENDSIZE Keyword ____ 9–12
9.1.11 FILE Keyword ____ 9–13
9.1.12 FORM Keyword ____ 9–13
9.1.13 INITIALSIZE Keyword ____ 9–14
9.1.14 IOSTAT Keyword ____ 9–14
9.1.15 KEY Keyword ____ 9–15
9.1.16 MAXREC Keyword ____ 9–16
9.1.17 NAME Keyword ____ 9–17
9.1.18 NOSPANBLOCKS Keyword ____ 9–17
9.1.19 ORGANIZATION Keyword ____ 9–17
9.1.20 READONLY Keyword ____ 9–18

9.1.21 RECL Keyword 9–18
9.1.22 RECORDSIZE Keyword 9–20
9.1.23 RECORDTYPE Keyword 9–20
9.1.24 SHARED Keyword 9–21
9.1.25 STATUS Keyword 9–21
9.1.26 TYPE Keyword 9–22
9.1.27 UNIT Keyword 9–22
9.1.28 USEROPEN Keyword 9–23

9.2 CLOSE STATEMENT 9–23

9.3 INQUIRE STATEMENT 9–24
9.3.1 ACCESS Specifier 9–25
9.3.2 BLANK Specifier 9–26
9.3.3 CARRIAGECONTROL Specifier 9–26
9.3.4 DIRECT Specifier 9–27
9.3.5 ERR Specifier 9–27
9.3.6 EXIST Specifier 9–27
9.3.7 FORM Specifier 9–28
9.3.8 FORMATTED Specifier 9–28
9.3.9 IOSTAT Specifier 9–28
9.3.10 KEYED Specifier 9–29
9.3.11 NAME Specifier 9–29
9.3.12 NAMED Specifier 9–30
9.3.13 NEXTREC Specifier 9–30
9.3.14 NUMBER Specifier 9–30
9.3.15 OPENED Specifier 9–31
9.3.16 ORGANIZATION Specifier 9–31
9.3.17 RECL Specifier 9–32
9.3.18 RECORDTYPE Specifier 9–32
9.3.19 SEQUENTIAL Specifier 9–33
9.3.20 UNFORMATTED Specifier 9–33

9.4 REWIND STATEMENT 9–34

9.5 BACKSPACE STATEMENT 9–35

9.6 ENDFILE STATEMENT 9–35

9.7 DELETE STATEMENT 9–36

9.8 UNLOCK STATEMENT 9–38

CHAPTER 10 COMPILER DIRECTIVES 10–1

10.1 COMPILER DIRECTIVE SYNTAX RULES 10–1

10.2 PARALLEL DIRECTIVES 10–2
10.2.1 CPAR$ CONTEXT_SHARED 10–3
10.2.2 CPAR$ CONTEXT_SHARED_ALL 10–3
10.2.3 CPAR$ DO_PARALLEL 10–4
10.2.4 CPAR$ LOCKON, CPAR$ LOCKOFF 10–5
10.2.5 CPAR$ PRIVATE 10–6
10.2.6 CPAR$ PRIVATE_ALL 10–7
10.2.7 CPAR$ SHARED 10–7
10.2.8 CPAR$ SHARED_ALL 10–8
10.2.9 Parallel Directive Examples 10–8

10.3 GENERAL DIRECTIVES 10–10
10.3.1 CDEC$ IDENT 10–10
10.3.2 CDEC$ PSECT 10–11
10.3.3 CDEC$ TITLE, CDEC$ SUBTITLE 10–13

APPENDIX A ADDITIONAL LANGUAGE FEATURES A–1

A.1 THE ENCODE AND DECODE STATEMENTS A–1

A.2 DEFINE FILE STATEMENT A–3

A.3 FIND STATEMENT A–5

A.4 PARAMETER STATEMENT A–6

A.5 OCTAL NOTATION FOR INTEGER CONSTANTS A–7

A.6 /NOF77 INTERPRETATION OF THE EXTERNAL STATEMENT A–8

APPENDIX B CHARACTER SETS B–1

B.1 FORTRAN CHARACTER SET B–1

B.2 ASCII CHARACTER SET B–2

B.3 RADIX–50 CONSTANTS AND CHARACTER SET B–4

APPENDIX C FORTRAN DATA REPRESENTATION C–1

C.1 INTEGER*2 REPRESENTATION C–1

C.2 INTEGER*4 REPRESENTATION C–2

C.3 LOGICAL*1 (BYTE) REPRESENTATION C–2

C.4 LOGICAL*2 AND LOGICAL*4 REPRESENTATION C–2

C.5 FLOATING-POINT REPRESENTATIONS C–3
C.5.1 REAL*4 (F_floating) C–4
C.5.2 REAL*8 (D_floating) C–5
C.5.3 REAL*8 (G_floating) C–6
C.5.4 REAL*16 (H_floating) C–7
C.5.5 COMPLEX*8 (F_floating) C–8
C.5.6 COMPLEX*16 (D_floating) C–8
C.5.7 COMPLEX*16 (G_floating) C–10

C.6 CHARACTER REPRESENTATION C–11

C.7 HOLLERITH REPRESENTATION C–11

APPENDIX D VAX FORTRAN LANGUAGE SUMMARY D–1

D.1 EXPRESSION OPERATORS D–1

D.2 STATEMENTS D–2

D.3 LIBRARY FUNCTIONS D–32

D.4 SYSTEM SUBROUTINE SUMMARY D–45
D.4.1 DATE Subroutine D–46
D.4.2 IDATE Subroutine D–47
D.4.3 ERRSNS Subroutine D–47
D.4.4 EXIT Subroutine D–48
D.4.5 SECNDS Subroutine D–48
D.4.6 TIME Subroutine D–49
D.4.7 RAN Subroutine D–50

D.5 BIT FUNCTIONS D–50
D.5.1 Bit Position D–51
D.5.2 Bit Function Arguments D–51
D.5.3 MVBITS Subroutine D–53

INDEX

EXAMPLES

6–1 Using Multiple Function Names 6–28

FIGURES

1–1 Required Order of Statements and Lines 1–3
1–2 FORTRAN Coding Form 1–10
1–3 Line Formatting Example 1–12
2–1 Array Storage 2–28
4–1 Equivalence of Substrings 4–18
4–2 Equivalence of Character Arrays 4–19

5–1 Control Transfers and Extended Range 5–10
5–2 Examples of Block IF Constructs 5–20

TABLES

1–1 Entities Identified by Symbolic Names 1–7
2–1 Data Type Storage Requirements 2–3
3–1 Conversion Rules for Assignment Statements 3–3
4–1 Equivalence of Array Storage 4–15
4–2 Equivalence of Arrays with Nonunity Lower Bounds 4–16
5–1 Nested DO Loops 5–7
6–1 Argument List Built-In Functions and Defaults 6–10
6–2 Types of User-Written Subprograms 6–12
6–3 Summary of Generic Intrinsic Function Names 6–27
7–1 List-Directed Default Output Formats 7–36
8–1 FORMAT Code Summary 8–6
8–2 Effect of Data Magnitude on G Format Conversions 8–23
8–3 Size Limit of Numeric Elements Using the A Field Descriptor 8–30
8–4 Default Field Descriptor Values 8–33
8–5 Carriage Control Characters 8–39
9–1 OPEN Statement Keyword Values 9–4
9–2 Record Size (RECL) Limits 9–19
9–3 Record Size (RECL) Default Values 9–19
10–1 Common Block Default Attributes and PSECT Modification 10–12
B–1 ASCII Character Set B–3
B–2 RADIX-50 Character Set with Comparative Values B–4
D–1 Expression Operators D–1
D–2 VAX FORTRAN Language D–3
D–3 VAX FORTRAN Intrinsic Functions D–33

Preface

This manual presents a complete description of the VAX FORTRAN language for VMS systems. It is designed as a reference manual, not as a tutorial document.

For detailed instructions on the features of the VAX FORTRAN compiler and its use, see the *VAX FORTRAN User Manual.*

Intended Audience

This manual is intended for programmers and students who have a basic understanding of the FORTRAN language. Readers do not need a detailed understanding of the VMS operating system, but some familiarity is helpful. For detailed information about the VMS system, refer to the VMS documentation set.

Structure of this Document

The documentation for VAX FORTRAN Version 5.0 is a major revision of the Version 4.0 documentation. The material is reorganized into two manuals. The *VAX FORTRAN User Manual* describes how to compile, link, execute, and debug VAX FORTRAN programs on the VMS system. It also describes special VAX FORTRAN features and system resources of interest to VAX FORTRAN programmers.

This manual presents the language-specific information. It is divided into ten chapters and four appendixes:

- Chapter 1 discusses VAX FORTRAN's relationship with FORTRAN standards, the elements of a source program, the character set, general coding rules, and compiler control statements.
- Chapter 2 describes the data types, data items, and expressions that can be used in VAX FORTRAN programs.
- Chapter 3 describes the assignment statement, which defines the values of data items.
- Chapter 4 describes specification statements, which are nonexecutable statements. Specification statements allocate and initialize data items and define various characteristics of symbolic names used in a program.
- Chapter 5 describes control statements, which specify when and where control transfers from one point in a program to another.
- Chapter 6 discusses subprograms (subroutines and functions), both those written by users and those supplied by VAX FORTRAN.
- Chapter 7 describes I/O (input/output) statements, which physically transfer data, both internally within memory and to and from output storage devices.
- Chapter 8 describes formatting statements, which are used together with formatted I/O statements.
- Chapter 9 describes auxiliary I/O statements that manage files.
- Chapter 10 describes compiler directives, which support directed decomposition and general-purpose functions.
- Appendix A describes some statements and language features that support programs written in older versions of FORTRAN.
- Appendix B summarizes the character sets supported by VAX FORTRAN.
- Appendix C shows how VAX FORTRAN data types are stored in memory.
- Appendix D summarizes VAX FORTRAN features: operators used in expressions, statements, intrinsic functions and their arguments, and system subroutines and bit manipulation functions.

Associated Documents

The following documents contain information directly related to the topic of this manual:

- *VAX FORTRAN User Manual*

  This manual describes how to perform basic operations using the VMS system and screen-display editor. The information about the VMS system and the editor should enable a programmer who is not acquainted with VMS to begin productive work on it.

- The VMS documentation set

  This set provides detailed information about features of the VMS operating system.

Conventions Used in this Document

The following syntactic conventions are used in this manual:

- Uppercase type is used in text to indicate VMS commands and command options.
- Lowercase letters are used in syntax specifications and examples to indicate variables; anything that is not a variable (for example, statement names and keywords) appears in uppercase.
- Brackets ([]) indicate optional elements within statements.
- Braces ({}) are used to enclose lists from which one element is to be chosen.
- Horizontal ellipses (...) indicate that the preceding items 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 denote special nonprinting characters:

Tab character	<TAB>
Space character	Δ

New and Expanded Language Features

VAX FORTRAN Version 5.0 provides the following new and expanded language features:

- Directed decomposition of DO-loops for parallel processing. Parallel processing is mediated by a group of compiler directives that are specifically called parallel directives. See Section 10.2.
- Several general-purpose functions: specifying the identification string in object modules, modifying some common block attributes, and listing title and subtitle. These functions are mediated by a group of compiler directives that are specifically called general directives. See Section 10.3.
- Support for descending ISAM (Indexed Sequential Access Mode) keys:
  - Expanded syntax for the KEY parameter in the OPEN statement. See Section 9.1.15.
  - Additional keywords in the key-field-value specifier in input /output statements. See Section 7.1.1.7.
- SIZEOF intrinsic function. This new function returns the number of bytes of storage used in a specified argument. See Table D–3.
- NWORKERS intrinsic function. This new function returns the number of processes executing a routine. See Table D–3.
- REWRITE operations on files that are open for direct access. See Section 7.4.
- UNLOCK operations of files with sequential organization. See Section 9.8.

See the *VAX FORTRAN User Manual* for a description of new compiler features available in VAX FORTRAN Version 5.0.

Chapter 1

Introduction to VAX FORTRAN

This chapter discusses VAX FORTRAN's relationship to FORTRAN standards, the elements of a VAX FORTRAN program, the character set, and general coding rules. It also presents compiler control statements.

VAX FORTRAN is based on the American National Standard FORTRAN–77 (ANSI X3.9-1978). It includes support for programs that conform to the previous standard (ANSI X3.9-1966). VAX FORTRAN also supports programs that conform to the International Standards Organization FORTRAN standard (ISO 1539-1980 (E)) because the ISO standard is the same as the ANSI standard.

VAX FORTRAN provides a number of extensions to the ANSI Standard:

- Compiler directives that support directed decomposition for parallel processing
- Compiler directives that perform several general-purpose functions
- Relative file organization
- Indexed file organization with two-directional keys
- 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

Extensions to the FORTRAN–77 standard appear in blue print in this manual.

VAX FORTRAN is also a compatible superset of PDP–11 FORTRAN–77. This means that existing PDP–11 FORTRAN–77 source programs will compile properly on the VAX FORTRAN compiler (see the *VAX FORTRAN User Manual*).

1.1 Elements of FORTRAN Source Programs

This section provides an overview of the makeup 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.

1.1.1 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 6 for detailed information on subprograms.

1.1.2 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.

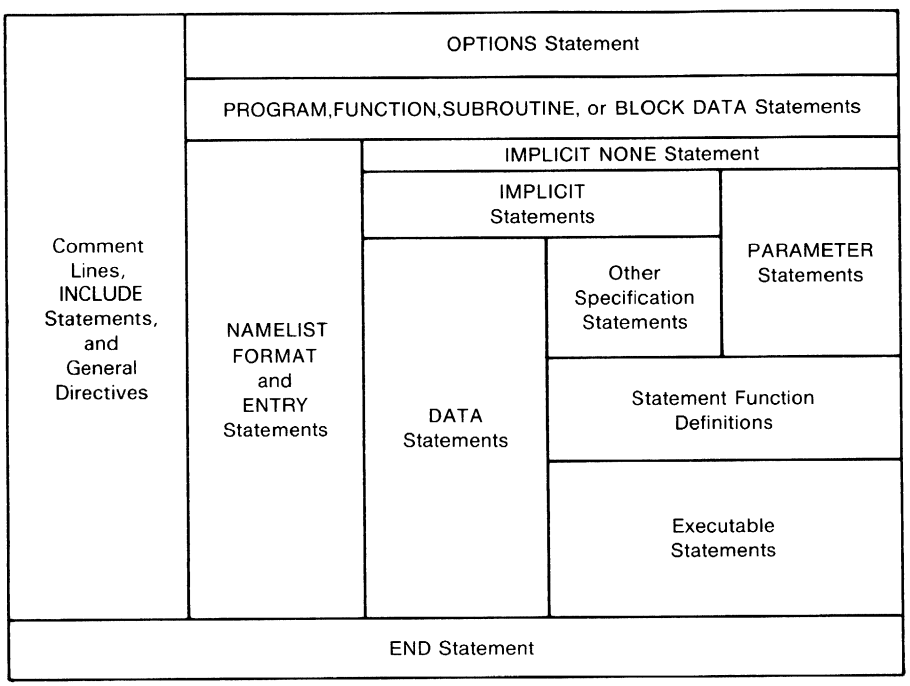
Statements are divided into physical sections called lines. A line is a string of up to 72 characters (optionally, 132; see Section 1.3.5). If a statement is too long to fit on one line, it can continue on one or more additional lines called continuation lines. A continuation line is identified by a continuation character in the sixth column of that line. (For further information on continuation characters, see Section 1.3.4.)

A statement label can identify a statement so that other statements can refer to it, either to get information or to transfer control. 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.

Order of Statements in a Program Unit

Figure 1–1 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, you can intersperse DATA statements with executable statements. On the other hand, horizontal lines indicate statement types that cannot be interspersed. For example, you cannot intersperse type declaration statements with executable statements.

Figure 1–1: Required Order of Statements and Lines



ZK-615-82

The following statements are the general directives that are included in the category with comment lines and INCLUDE statements in Figure 1–1:

CDEC$ IDENT
CDEC$ SUBTITLE
CDEC$ TITLE
CDEC$ PSECT

The following statements are included in the category of executable statements in Figure 1–1:

ACCEPT
ASSIGN
Assignment statements
BACKSPACE
CALL
CLOSE
CONTINUE
CPAR$ DO_PARALLEL
CPAR$ LOCKOFF
CPAR$ LOCKON
DELETE
DO and END DO
ELSE
END
ENDFILE
FIND
GO TO (normal, computed, assigned)
IF (arithmetic, logical, block) and END IF
INQUIRE
OPEN
PAUSE
PRINT
READ
RETURN
REWIND
REWRITE
STOP
TYPE
UNLOCK
WRITE

The following statements are included in the category of "other specification statements" in Figure 1–1:

COMMON	EQUIVALENCE
CPAR$ CONTEXT_SHARED	EXTERNAL
CPAR$ CONTEXT_SHARED_ALL	INTRINSIC
CPAR$ PRIVATE	RECORD
CPAR$ PRIVATE_ALL	SAVE
CPAR$ SHARED	Structure declarations[1]
CPAR$ SHARED_ALL	Type declarations
DICTIONARY	VOLATILE
DIMENSION	

[1]The statements STRUCTURE and END STRUCTURE, UNION and END UNION, and MAP and END MAP are included in the "other specification statements" 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.

1.1.3 Symbolic Names

Symbolic names identify entities within a FORTRAN program unit. These entities are listed in Table 1–1.

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. (FORTRAN–77 limits the length of a symbolic name to six characters.)

Examples

The following examples demonstrate valid and invalid symbolic names and explain why the invalid ones are not valid:

Valid	
NUMBER	
FIND_IT	
X	
Invalid	**Explanation**
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 DIGITAL-supplied software components. To avoid name conflicts, do not define any symbolic names in your program that contain dollar signs.

Symbolic names cannot identify more than one entity in the same program unit—except when they identify common blocks, records, structures, structure fields, and either arrays or variables. For example, in the following valid statements, X is the name of a common block, a structure, a structure field, and a variable:

```
COMMON /X/ I, J
STRUCTURE /X/
INTEGER X
END STRUCTURE
REAL X
```

Section 4.15.1 and Section 4.15.2 provide more information about structure and field names.

In an executable program having 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 program
- Block data subprograms
- Function entry points
- Subroutine entry points

For example, 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 1–1 lists those entities that can be given a symbolic name. It also indicates whether the entities can be given a data type. Sections 2.2.2.1 and 2.2.2.2 discuss how to specify the data type of a symbolic name.

Table 1–1: 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

1.1.4 Comments

Comments are documentation aids that do not affect program processing in any way. They can be freely used to describe the actions of a program, identify program sections and processes, and provide greater ease in reading a source program listing.

Three different characters identify comments:

- The letter C — except when C begins a compiler directive
- An asterisk (*)
- An exclamation point (!)

When the letter C or an asterisk appears in the first column of a source line, it identifies the line as a comment. An exclamation point in the first column, or anywhere in the statement portion of a source line, identifies the remainder of that line as a comment.

An all-blank line is also treated as a comment line.

Section 10.2.9 provides an example of when the letter C in column 1 begins a compiler directive instead of a comment.

1.2 Character Set

VAX FORTRAN supports the following character set:

- All uppercase and lowercase letters (A through Z, a through z)
- The numerals 0 through 9
- The following special characters:

Character	Name	Character	Name
Δ 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

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. If nonprinting characters appear anywhere in a FORTRAN source statement, they appear as question marks in the compilation source listing.

The control character <NEWLINE> (<LF> or 'A' X) is not supported in VAX FORTRAN source records. VAX FORTRAN uses <NEWLINE> to separate successive FORTRAN source lines. If you must manipulate this character, use the CHAR(10) function reference, which is allowed in all compile-time character expressions (see Section 6.3.2.1).

Except in character and Hollerith constants, the compiler makes no distinction between uppercase and lowercase letters.

1.3 Coding Rules

Coding rules specify the structure of lines in VAX FORTRAN source code. A line has four fields: the statement label, continuation indicator, statement, and sequence number. Rules affecting individual fields are described in Sections 1.3.3 through 1.3.6.

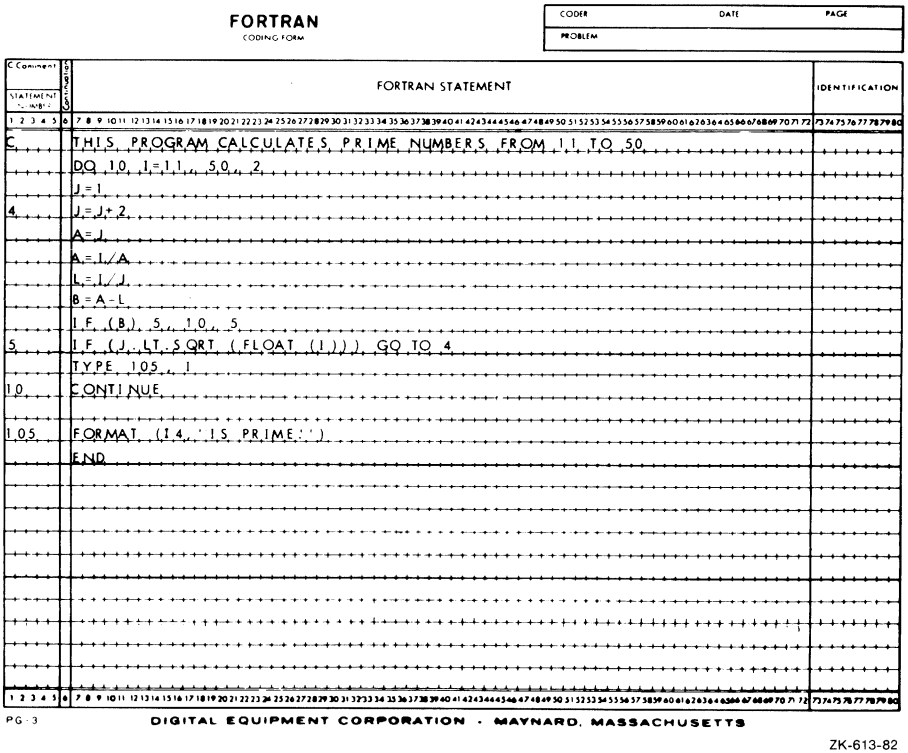
There are two ways to code a line: by fixed format or tab format. The fixed-format method is convenient when you punch cards or use a coding form. The tab-format method is convenient when you enter lines at a terminal with a text editor.

1.3.1 Fixed-Format Lines

As shown in Figure 1–2, a FORTRAN line is divided into fields for statement labels, continuation indicators, statement text, and sequence numbers. Each column represents a single character.

To enter an item in a field, enter it in the columns in the coding form, as follows:

Figure 1–2: FORTRAN Coding Form


FORTRAN
CODING FORM
CODER DATE PAGE
PROBLEM
C Comment STATEMENT NUMBER CONTINUATION
FORTRAN STATEMENT
IDENTIFICATION
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=I/A
L=I/J
B=A-L
IF (B),5,,10,,5
5 IF (J.LT.SQRT (FLOAT (I))) GO TO 4
TYPE 105,,I
10 CONTINUE
105 FORMAT (I4,' IS PRIME.')
END
PG-3
DIGITAL EQUIPMENT CORPORATION · MAYNARD, MASSACHUSETTS
ZK-613-82

Field	Column
Statement label	1 through 5
Continuation indicator	6
Statement	7 through 72 (optionally, to 132)
Sequence number	73 through 80

1.3.2 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 1–3 illustrates FORTRAN lines coded using tab formatting and the equivalent lines with fixed formatting.

Figure 1–3: Line Formatting Example

Format Using TAB Character

C (TAB) FIRST VALUE

10 (TAB) I = J + 5*K +

(TAB) 1 L*M

(TAB) IVAL = I+2

Character-per-Column Format

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			
1	0					I		=		J		+		5	*	K		+	
					1		L	*	M										
						I	V	A	L		=		I	+	2				

ZK-614-82

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 typing 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. 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 the VAX FORTRAN compiler's interpretation of the tab character. The compiler treats the tab character in a statement field the same way as it treats 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.

1.3.3 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.

Labeled FORMAT and labeled executable statements are the only statements that can be referred to by other statements (see Section 1.1.2). 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.

The statement label field of a continuation line must be blank—except in the case of a debugging statement.

1.3.3.1 Comment Indicator

The letter C (except when beginning a compiler directive); an asterisk (*); or an exclamation point (!) in column 1 indicates that the line is a comment. The compiler prints that line in the source program listing and then ignores it. 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.

See Chapter 10 for a description of when the letter C in column 1 begins a compiler directive instead of a comment.

1.3.3.2 Debugging Statement Indicator

The letter D in column 1 designates 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 debugging statements either as source text to be compiled or as comments, depending on the setting of the /D_LINES qualifier on the FORTRAN command. If you specify the /D_LINES qualifier, debugging statements are compiled as a part of the source program. If you do not specify the /D_LINES qualifier, debugging statements are treated as comments. See the *VAX FORTRAN User Manual* for more information on the /D_LINES qualifier.

1.3.4 Continuation Indicator Field

A continuation indicator is any character, except a zero or a 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 a 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 lines, or between successive continuation lines.

1.3.5 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 line, the statement field is extended to position 132. Any text beyond that position generates a fatal error and causes immediate termination of the compilation. See the *VAX FORTRAN User Manual* for more information on the /EXTEND_SOURCE qualifier.

1.3.6 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 line, a sequence number field does not exist; the statement field is extended to position 132. See the *VAX FORTRAN User Manual* for more information on the /EXTEND_SOURCE qualifier.

1.4 Compilation Control Statements

In addition to qualifiers on the FORTRAN command line, several statements used in the body of a VAX FORTRAN program also influence compilation:

- DICTIONARY—extracts records from the Common Data Dictionary (CDD) and converts them into VAX FORTRAN records.
- INCLUDE—incorporates external source code into programs.
- OPTIONS—establishes compiler qualifiers otherwise specified on the FORTRAN command line; overrides command line qualifiers if a conflict occurs between OPTIONS statement qualifiers and command line qualifiers.

1.4.1 DICTIONARY Statement

The DICTIONARY statement incorporates VAX Common Data Dictionary data definitions into the current VAX FORTRAN source file during compilation. It can occur anywhere in a VAX FORTRAN source file that a specification statement can occur (see Chapter 4).

The DICTIONARY statement takes the following form:

```
DICTIONARY 'cdd-path [/[NO]LIST]'
```

cdd-path

Is interpreted as the *full* or *relative* pathname of a CDD object.

/[NO]LIST

Controls whether the source code representation of the resulting structure declaration is listed in a compilation source listing. The default is /NOLIST.

Syntax Rules and Behavior

There are two types of CDD pathname: full and relative. Their formation must conform to the rules for forming VAX CDD pathnames.

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 comes into existence when a default directory is established with a logical name.

Examples

In the following example, the logical name definition specifies the beginning of the CDD pathname; thus, a relative pathname specifies the remainder of the path to the record definition:

```
$ DEFINE CDD$DEFAULT CDD$TOP.FOR
```

The following examples illustrate how a CDD pathname beginning with CDD$TOP overrides the default CDD pathname. Consider a record with the pathname CDD$TOP.SALES.JONES.SALARY. If you defined CDD$DEFAULT to be CDD$TOP.SALES.JONES, you could then specify a relative pathname:

```
DICTIONARY 'SALARY'
```

Alternatively, you could specify a full pathname:

```
DICTIONARY 'CDD$TOP.SALES.JONES.SALARY'
```

See the *VAX Common Data Dictionary Utilities Manual* for further details.

1.4.2 INCLUDE Statement

The INCLUDE statement directs the compiler to stop reading statements from the current file and read 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.

The INCLUDE statement takes one of the following forms:

```
INCLUDE  '[text-lib] (module-name)[/[NO]LIST]'

INCLUDE  'file-spec[/[NO]LIST]'
```

text-lib
Is a character string that specifies the text library to be searched. Its form must be acceptable to the operating system, as described in the *VAX FORTRAN User Manual.*

file-spec
Is a character string that specifies the file to be included. The form of the file-spec must be acceptable to the operating system, as described in the *VAX FORTRAN User Manual.*

module-name
Is the name of the text module, located in a text library, that is to be included. 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
Specifies whether the incorporated code is to appear in the compilation source listing. In the listing, a number precedes each incorporated statement. The number indicates the "include" nesting depth of the code.

The default is /NOLIST.

Syntax Rules and Behavior

The INCLUDE statement and included files have the following rules:

- An included file or module cannot begin with a continuation line. Each VAX FORTRAN statement must be completely contained within a single file or module.
- An included file or module can contain an INCLUDE statement.
- The INCLUDE statement can appear anywhere within a program unit.
- Any VAX 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 Figure 1-1.

Example

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
```

COMMON.FOR File

```
PARAMETER (M=100)
COMMON X(M),Y(M)
```

1.4.3 OPTIONS Statement

The OPTIONS statement overrides or confirms the FORTRAN command line qualifiers in effect for a program unit. It takes the following form:

```
OPTIONS qualifier[qualifier...]
```

qualifier

Is one of the following:

```
/[NO]G_FLOATING
/[NO]I4
/[NO]F77
          ALL
          [NO]OVERFLOW
/CHECK =  [NO]BOUNDS
          [NO]UNDERFLOW
          NONE
/NOCHECK
/[NO]EXTEND_SOURCE
```

Syntax Rules and Behavior

The OPTIONS statement must be the first statement in a program unit, preceding the PROGRAM, SUBROUTINE, FUNCTION, and BLOCK DATA statements.

OPTIONS statement qualifiers have the same syntax and abbreviations as the FORTRAN command line qualifiers, as described in the *VAX FORTRAN User Manual*.

The OPTIONS qualifiers override FORTRAN command line qualifiers, but 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.

Example

In the following example, the check and extend_source options do not remain in effect across program unit boundaries. The first OPTIONS statement specifies that only the program unit immediately following it 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 second OPTIONS statement specifies that only the program unit following it is to be compiled with the G_floating option.

```
OPTIONS /CHECK/EXTEND_SOURCE
.
.
.
END
OPTIONS /G_FLOATING
.
.
.
```

Chapter 2

Data Types, Data Items, and Expressions

This chapter contains information on the following topics:

- Data types—integer, real, complex, logical, character, and BYTE (Section 2.1)
- Data items—constants, variables, arrays, character substrings, and records (Section 2.2)
- Expressions—arithmetic, character, relational, and logical (Section 2.3)

2.1 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 following data types are available in VAX FORTRAN:

- 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, but 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 but 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 printable ASCII characters.
- BYTE— a one-byte storage location that is equivalent to LOGICAL*1.

See Appendix C for descriptions of the VAX hardware representations of these data types. See Section 4.4 for descriptions of data type declaration statements.

2.1.1 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 2–1 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 2–1: 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
COMPLEX	8
COMPLEX*8	8
COMPLEX*16	16
DOUBLE COMPLEX	16
CHARACTER*len	len[3]
CHARACTER*(*)	

[1]BYTE and LOGICAL*1 are equivalent. See Section C.3 for information on the range of values that can be stored in them.

[2]Either two or four bytes are allocated, depending on the setting of the [NO]I4 qualifier on the FORTRAN command line. 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 Section 6.1.1.3 and the *VAX FORTRAN User Manual*).

2.1.2 VAX Implementations of REAL*8

The REAL*8 (and thus, COMPLEX*16) data type has two implementations on a VAX computer: D_floating and G_floating. The G_floating implementation offers a greater range but is less precise, having a smaller number of significant digits. D_floating is the default implementation of REAL*8. You can select G_floating by using the OPTIONS statement or the /G_FLOATING qualifier on the FORTRAN command line.

Some VAX processors emulate floating-point data types rather than executing them in hardware or microcoded instruction. Processing time with software emulation is much slower. Thus, you should be aware of which data types are emulated on your system and choose them (especially REAL*8) with this information in mind.

See Sections 2.2.1.2, C.5, C.5.2, and C.5.3 for more detailed information on the two implementations of the REAL*8 data type.

2.2 Data Items

VAX FORTRAN statements use the following data items:

- 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.

2.2.1 Constants

A constant is a data item with a fixed value that 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 2.2.1.4 and 2.2.1.7).

All eight types of constants are *scalar references* in that they resolve into single, typed data items. (See Section 2.2.6 for more information on scalar references.)

2.2.1.1 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. Integer constants take the following form:

```
snn
```

s
Is an optional sign.

nn
Is a string of decimal digits. Any leading zeros are ignored.

Syntax Rules

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

The following examples demonstrate valid and invalid integer constants and explain why the invalid ones are not valid:

Valid	
0	
-127	
+32123	
Invalid	**Explanation**
99999999999	Number too large
3.14	Decimal point not allowed
32,767	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 be used to assign signed and unsigned values to BYTE, LOGICAL*1, LOGICAL*2, and INTEGER*2 data. BYTE and LOGICAL*1 can contain signed integers with a range of -128 to 127 or unsigned integers with a range of 0 to 255. LOGICAL*2 and INTEGER*2 can contain signed integers with a range of -32768 to 32767 or unsigned integers with a range of 0 to 65535.

The following table illustrates VAX FORTRAN assignments to different data and lists the integer and hexadecimal values in the data:

VAX FORTRAN Assignment	Integer Value in the Data	Hexadecimal Value in the Data
BYTE X		
X = -128	-128	'80'X
X = 127	127	'7F'X
X = 255	-1	'FF'X
LOGICAL*1 X		
X = -128	-128	'80'X
X = 127	127	'7F'X
X = 255	-1	'FF'X
LOGICAL*2 X		
X = -32768	-32768	'8000'X
X = 32767	32767	'7FFF'X
X = 65535	-1	'FFFF'X
INTEGER*2 X		
X = -32768	-32768	'8000'X
X = 32767	32767	'7FFF'X
X = 65535	-1	'FFFF'X

Integer constants can also be specified in octal form, as described in Sections 2.2.1.4 and A.5.

2.2.1.2 Real Constants

A real constant is a number written with a decimal point, exponent, or both. The constant can be positive, zero, or negative. It can have single precision (REAL*4), double precision (REAL*8), or quad precision (REAL*16).

REAL*4 (REAL) Constants

A REAL*4 constant can be any one of the following:

- Basic real constant
- Basic real constant followed by a decimal exponent
- Integer constant followed by a decimal exponent

A basic real constant takes one of the following forms:

```
  s.nn
snn.nn
snn.
```

s
Is an optional sign.

nn
Is a string of decimal digits. A decimal point can appear anywhere in the string.

A decimal exponent takes the following form:

```
Esnn
```

s
Is an optional sign.

nn
Is a string of decimal digits.

Syntax Rules and Behavior

A REAL*4 constant occupies four bytes of VAX storage. It is interpreted as a real number with a degree of precision that is typically seven decimal digits (see Sections C.5 and C.5.1.

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.

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 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 can only contain the numerals 0 through 9, algebraic signs, a decimal point, and the letter E.

When the letter E appears in a REAL*4 constant, an integer constant exponent field must follow. The exponent field cannot be omitted, but it can 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

The following examples demonstrate valid and invalid REAL*4 constants and explain why the invalid ones are not valid:

Valid
3.14159
621712.
-.00127
+5.0E3
2E-3

Invalid	Explanation
1,234,567.	Commas not allowed
325E-45	Too small
-47.E47	Too large
100	Decimal point missing - this is a valid integer constant
$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. Its decimal exponent takes the following form:

```
Dsnnn
```

s
Is an optional sign.

nnn
Is a string of decimal digits.

Syntax Rules and Behavior

A REAL*8 constant occupies eight bytes of VAX storage. The number of digits that precede the exponent is unlimited.

Both D_floating and G_floating implementations of REAL*8 have the same syntax and storage requirements. However, they differ in the number of significant digits and exponential range. D_floating implementation typically has 16 (leftmost) significant numbers and a two-digit exponent. On the other hand, G_floating typically has 15 (leftmost) significant digits and a three-digit exponent. See Sections C.5, C.5.2, and C.5.3.

D_floating is the default implementation. To select G_floating implementation, you must include the /G_FLOATING qualifier on the FORTRAN command line.

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.

A REAL*8 constant can only contain the numerals 0 through 9, algebraic signs, a decimal point, and the letter D.

An integer constant exponent field must follow the letter D. The exponent field cannot be omitted, but it can 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

The following examples demonstrate valid and invalid D_floating and G_floating REAL*8 constants and explain why the invalid ones are not valid:

D_floating REAL*8 Constants

Valid	
1234567890D+5	
+2.71828182846182D00	
-72.5D-15	
1D0	
Invalid	**Explanation**
1234567890D45	Too large
1234567890.0D-89	Too small
+2.7182812846182	No Dsnnn present; this is a valid single-precision constant

G_floating REAL*8 Constants	
Valid	
123456789.D0	
+2.34567890123D-5	
-1D+300	
Invalid	**Explanation**
123456789.D400	Too large
123456789.D-400	Too small

REAL*16 Constants

A REAL*16 constant is a basic real constant or an integer constant followed by a decimal exponent. Its decimal exponent takes the following form:

```
Qsnnnn
```

s
Is an optional sign.

nnnn
Is a string of decimal digits.

Syntax Rules and Behavior

A REAL*16 constant occupies 16 bytes of VAX storage. The number of digits that precede the exponent is unlimited; however, typically only the leftmost 33 digits are significant (see Sections C.5 and C.5.4).

A minus sign must appear before a negative REAL*16 constant, but a plus sign is optional before a positive constant. Similarly, a minus sign is required between the letter Q and a negative exponent, but a plus sign is optional between the letter Q and a positive exponent.

A REAL*16 constant can only contain the numerals 0 through 9, algebraic signs, a decimal point, and the letter Q.

An integer constant exponent field must follow the letter Q. The exponent field cannot be omitted, but 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

The following examples demonstrate valid and invalid REAL*16 constants and explain why the invalid ones are not valid:

Valid

```
1234567890Q4000
-1.23Q-400
+2.72Q0
```

Invalid	Explanation
1.Q5000	Too large
1.Q-5000	Too small

2.2.1.3 Complex Constants

A complex constant is 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; the second constant represents the imaginary part.

VAX FORTRAN supports COMPLEX*8 and COMPLEX*16 complex constants.

COMPLEX*8 (COMPLEX) Constants

A COMPLEX*8 constant is a pair of integer or REAL*4 constants that represents a complex number. It takes the following form:

```
(c,c)
```

c
Is an integer or REAL*4 constant.

The parentheses and comma are required parts of the constant. (See Section 2.2.1.2 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.5 and C.5.5).

Examples

The following examples demonstrate valid and invalid COMPLEX*8 constants and explain why the invalid ones are not valid:

Valid

```
(1.7039,-1.70391)
(+12739E3,0.)
(1,2)
```

Invalid	Explanation
`(1.23,)`	Missing second REAL constant
`(1.0,1.0Q0)`	REAL*16 constant not allowed

COMPLEX*16 (DOUBLE COMPLEX) Constants

A COMPLEX*16 constant is a pair of constants that represents 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 takes the following form:

```
(c,c)
```

c
Is an integer, a REAL*4, or a REAL*8 constant. (One of the pair must be a REAL*8 constant.)

A COMPLEX*16 constant occupies 16 bytes of VAX storage and is interpreted as a complex number (see Sections C.5, C.5.6, and C.5.7).

The parentheses and the comma are required parts of the constant.

Syntax rules for REAL*8 constants also apply to the REAL*8 portion of COMPLEX*16 constants.

Examples

The following examples demonstrate valid and invalid COMPLEX*16 constants and explain why the invalid examples are not valid:

Valid

```
(1.7039D0,-1.7039D0)

(+12739D3,0.D0)
```

Invalid	Explanation
`(1.23D0)`	Second constant missing
`(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; they are valid for G_floating implementation of REAL*8

2.2.1.4 Octal and Hexadecimal Constants

Octal and hexadecimal constants are alternative ways to represent numeric constants. They can appear wherever numeric constants are allowed.

An octal constant is a string of octal digits enclosed by apostrophes and followed by the letter O. It takes the following form:

```
'c1c2c3...cn'O
```

cn
Is a digit with a range of 0 to 7.

A hexadecimal constant is a string of digits enclosed by apostrophes and followed by the alphabetic character X. It takes the following form:

```
'c1c2c3...cn'X
```

cn
Is a digit in the range of 0 to 9, or an uppercase or lowercase letter in the range of 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

The following examples demonstrate valid and invalid octal and hexadecimal constants and explain why the invalid ones are not valid:

Octal Constants	
Valid	
'07737'O	
'1'O	
Invalid	**Explanation**
'7782'O	The character 8 is invalid
7772'O	No initial apostrophe
'0737'	No O after second apostrophe

Hexadecimal Constants	
Valid	
'AF9730'X	
'FFABC'X	
Invalid	**Explanation**
'999.'X	Invalid character
'F9X	No apostrophe before X

Data-Typing Octal and Hexadecimal Constants

Octal and hexadecimal constants have no data type until they are used. When used, they assume a data type based on their use.

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 (in bytes)
`INTEGER*2 ICOUNT`		
`REAL*8 DOUBLE`		
`RAPHA = '99AF2'X`	REAL*4	4
`JCOUNT = ICOUNT + '777'O`	INTEGER*2	2
`DOUBLE = 'FFF99A'X`	REAL*8	8
`IF (N .EQ. '123'O) 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 (in bytes)
`Y(IX) = Y('15'O) + 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 (in bytes)
`CALL APAC('34BC2'X)`	None	4

When the constant is used in any other context, an INTEGER*4 data type is assumed (or an INTEGER*2 data type if /NOI4 is in effect). For example:

Statement	Data Type of Constant	Length of Constant (in bytes)
`IF ('AF77'X) 1,2,3`	INTEGER*4	4
`I = '7777'O - 'A39'X`	INTEGER*4	4
`J = .NOT. '73777'O`	INTEGER*4	4

An octal or hexadecimal constant specifies up to 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 2–1 lists the number of bytes that each data type requires.

2.2.1.5 Logical Constants

A logical constant specifies a logical value, true or false. Thus, only the following two logical constants are possible:

```
.TRUE.
.FALSE.
```

Both delimiting periods are required.

2.2.1.6 Character Constants

A character constant is a string of printable ASCII characters enclosed by apostrophes. It takes the following form:

```
'c1c2c3...cn'
```

cn
Is a printable ASCII character.

Syntax Rules and Behavior

Both delimiting apostrophes are required.

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 of 1 to 2000.

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 Section 2.2.1.7).

Examples

The following examples demonstrate valid and invalid character constants and explain why the invalid ones are not valid:

Valid

```
'WHAT?'
'TODAY''S DATE IS:'
'HE SAID, "HELLO"'
```

Invalid	Explanation
`'HEADINGS`	No trailing apostrophe
`' '`	Character constant must contain at least one character
`"NOW/OR NEVER"`	Quotation marks cannot take the place of apostrophes

2.2.1.7 Hollerith Constants

A Hollerith constant is a string of printable ASCII characters preceded by a character count and the letter H. It takes the following form:

```
nHc1c2c3...cn
```

n

Is an unsigned, nonzero integer constant stating the number of characters in the string (including spaces and tabs).

cn

Is a printable ASCII character.

Hollerith constants are strings of 1 to 2000 characters. They are stored as byte strings, one character per byte.

Examples

The following examples demonstrate valid and invalid hollerith constants and explain why the invalid ones are not valid:

Valid	
16HTODAY'S DATE IS:	
1HB	
Invalid	**Explanation**
3HABCD	Wrong number of characters
0H	Hollerith constants must contain at least one character

Data-Typing Hollerith Constants

Hollerith constants have no data type. They assume a numeric data type based on the way they are used. Hollerith constants cannot assume a character data type and cannot be used where a character value is expected.

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 (in bytes)
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 (in bytes)
`Y(IX) = Y(1HA) + 3.`	INTEGER*4	4

When the constant is used as an actual argument, no data type is assumed.

Statement	Data Type of Constant	Length of Constant (in bytes)
`CALL APAC(9HABCDEFGHI)`	None	9

When the constant is used in any other context, an INTEGER*4 data type is assumed (or an INTEGER*2 data type if /NOI4 is in effect). For example:

Statement	Data Type of Constant	Length of Constant (in bytes)
`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 2–1 (in Section 2.1.1) lists the number of characters required for each data type. Each character occupies one byte of storage.

2.2.2 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 its value at any point in a program by assigning a new value to it. (See Section 1.1.3 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 contains, its precision, and its storage requirements. When data of any type is assigned to a variable, it is converted, if necessary, to the data type of the variable.

The following statements and rules establish the data type of a variable:

- Type declaration statements
- IMPLICIT statements
- Predefined typing rules

All types of variables are scalar references in that they resolve into single, typed data items. (Section 2.2.6 discusses scalar references.)

Associating Variables

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 or EQUIVALENCE statements; MAP declarations (within structure declaration blocks) or actual arguments and dummy arguments in subprogram references.

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. This occurs because a variable is defined only if the storage associated with it contains data of the same type as the name.

A variable can be defined before program execution by a DATA statement or during execution by an assignment or input statement.

2.2.2.1 Data Type by Specification

Data type declaration statements explicitly specify the data type of variables (see Section 4.4). For example, the following statements associate VAR1 with an 8-byte complex-data storage location, and VAR2 with an 8-byte double-precision storage location:

```
COMPLEX VAR1
DOUBLE PRECISION VAR2
```

You can explicitly specify the data type of a variable only once.

An explicit data type specification takes precedence over the type specified by an IMPLICIT statement. The data type specified by an IMPLICIT statement associates with a variable only when an explicit specification is absent. Thus, in the absence of an explicit specification, any variable with a name that begins with the letter in the range specified in the IMPLICIT statement becomes the data type of the variable.

Character type declaration statements (see Sections 4.4 and 4.4.2) specify that given variables represent character values with the length specified. For example, the following statements associate the variable names INLINE, NAME, and NUMBER with storage locations containing character data of lengths 72, 12, and 9, respectively:

```
CHARACTER*72 INLINE
CHARACTER NAME*12, NUMBER*9
```

In single subprograms, passed-length character arguments process character strings with 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 Section 6.1.1.3 and the *VAX FORTRAN User Manual*.)

2.2.2.2 Data Type by Implication

In the absence of either IMPLICIT statements or explicit data type declarations, 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

2.2.3 Arrays

An array is a group of contiguous storage locations associated with a single symbolic name, the *array name*. The individual storage locations, *array elements*, are referenced by a subscript appended to the array name.

An array can have from one to seven dimensions. For example, a column of figures is a one-dimensional array. A table with 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 and column numbers. A table of figures that covers several pages is a three-dimensional array. To locate a value in this array, you must specify the page, row, and column numbers.

The following VAX FORTRAN statements establish arrays:

- Data type declaration (see Section 4.4)
- DIMENSION (see Section 4.5)
- COMMON (see Section 4.2)

These statements contain array declarators that define the name of the array, the number of dimensions in the array, and the number of elements in each dimension.

Associating Arrays

Two or more arrays are associated when each one is associated with the same storage location. They are partially associated when part of the storage associated with one array is the same as part or all of the storage associated with another array.

Association and partial association occur when you use COMMON or EQUIVALENCE statements; MAP declarations (within structure declaration blocks); or actual arguments and dummy arguments in subprogram references.

If arrays with different data types are associated (or partially associated) with the same storage location, and the value of one array is defined (for example, by assignment), the value of the other array becomes undefined. This happens because an element of an array is considered defined only if the storage associated with it contains data of the same type as the array name (see Section 2.2.3.3).

The DATA statement defines an array element or an entire array before program execution. During program execution, array elements are defined by an assignment or input statement, and entire arrays are defined by input statements.

2.2.3.1 Array Declarators

An array declarator specifies the symbolic name that identifies an array within a program unit and indicates the properties of that array. It takes the following form:

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

a

Is the symbolic name of the array. (Section 1.1.3 gives the form of a symbolic name.)

d

Is a dimension declarator that can specify both a lower bound and an upper bound:

```
[dl:]du
```

dl

Is the lower bound of the dimension.

du

Is the upper bound of the dimension. An asterisk (*) can be used as an upper bound, but only for the last dimension of a dummy argument. An asterisk marks the declarator as an assumed-size array declarator (see Section 6.1.1.2).

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.

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

Do not use array references and references to user-defined functions 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 6.1.1.1 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.

2.2.3.2 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.

Subscript array references are scalar references in that they resolve into single, typed data items. (Section 2.2.6 describes this terminology.)

A subscript takes the following form:

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

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).

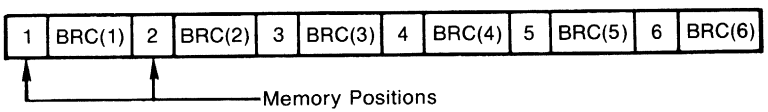
Each subscript can be any valid arithmetic expression. However, non-integer subscript expressions are converted to integers before use (any fractional parts are truncated).

2.2.3.3 Arrangement of Array Elements in Storage

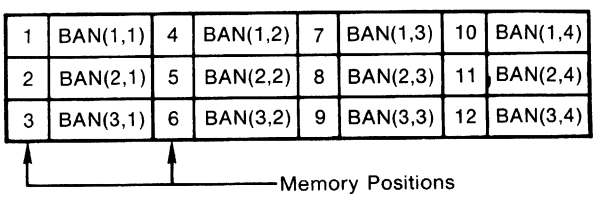
As discussed earlier in this section, you can think of the dimensions of an array as pages, rows, and columns. However, FORTRAN actually stores arrays in memory as a linear sequence of values. A one-dimensional array is stored with its first element in the first storage location and its last element 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 2–1 shows array storage in one, two, and three dimensions.

Figure 2–1: Array Storage

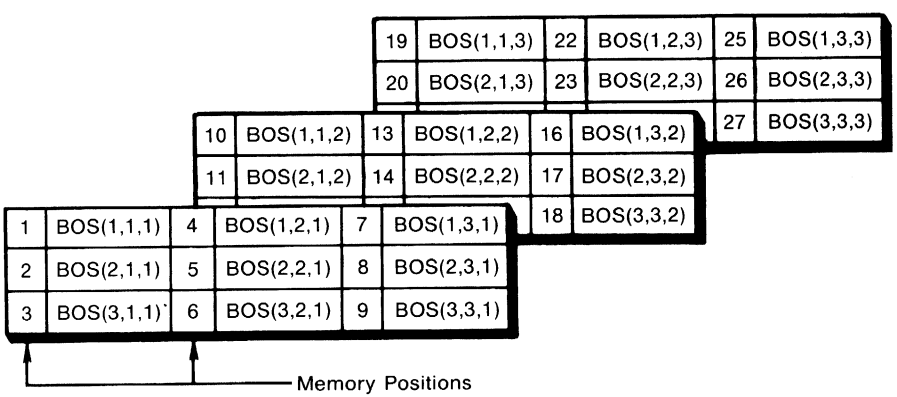
One-Dimensional Array BRC (6)



Two-Dimensional Array BAN (3,4)



Three-Dimensional Array BOS (3,3,3)



ZK-616-82

2.2.3.4 Data Type of an Array

The data type of an array is specified in the same way as the data type of a variable—implicitly by the initial letter of the name, or explicitly by a data type declaration statement (see Sections 2.2.2.1 and 2.2.2.2).

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.

2.2.3.5 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
- DATA
- EQUIVALENCE
- NAMELIST
- SAVE
- I/O
- Data type declaration

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. Using unsubscripted array names is not permitted in all other types of statements.

2.2.3.6 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 name, as subprogram arguments. The bounds may also be given in a common block. See Section 6.1.1.1 for more information.

2.2.3.7 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 6.1.1.2 describes the rules governing the dimensions that are assumed.

2.2.4 Character Substrings

A character substring is a contiguous segment of a character variable or character array element. It takes one of the following forms:

```
v([e1]:[e2])
a(s[,s]...) ([e1]:[e2])
```

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) .AND. (e1 .LE. e2) .AND. (e2 .LE. len)
```

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).

2.2.5 Records

A record is an aggregate entity containing one or more elements. (Record elements are also called fields or components.) You can use records when you need to declare and operate on multifield data structures in your VAX FORTRAN programs. You can also access records in the VAX Common Data Dictionary (CDD) and use them in your programs.

NOTE

Do not confuse a VAX FORTRAN record with an RMS I/O record. VAX FORTRAN records are named data entities with one or more fields that you create in your program.

A record is similar to an array in that they both contain one or more elements. However, a record differs from an array in the following respects:

- Unlike arrays, which are defined by a single declaration statement, creating a record is a two-step process:
  1. Defining the form of the record with a multistatement *structure declaration*.
  2. Declaring the record as an entity with a symbolic name, thus establishing its structure in memory. More than one RECORD statement can refer to a given structure.
- Unlike arrays, whose data elements must have the same data type, records can have fields with different data types. Because records have heterogeneous data elements, they are not typed as arrays are. Record fields can be operated on individually or collectively.

- Unlike array elements, each element of a record can be named. References to a record element consist of both the name of the record containing the element and the name of the desired element.

Structure Declaration Blocks

A structure declaration block is a named group of statements that define the form of a record.

To establish a structure declaration in memory, its name must be specified in a RECORD statement.

A structure declaration block includes one or more of the following items:

- *Data Type Declarations* (variables or arrays): Data type declarations in structure declarations have the form of normal VAX FORTRAN data type 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 using 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, within a structure declaration, a RECORD statement naming the other structure. 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 (nested 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 similar to using EQUIVALENCE statements to give names to variables and arrays. In other languages, it is called a "variant record" capability.

- *Unnamed Fields*: Unnamed fields can be declared in a structure by specifying %FILL in place of an actual field name. This mechanism can be used to generate space in a record for purposes such as alignment.

  Unnamed fields cannot be initialized. For example, the following field declaration is invalid and generates an error message:

```
INTEGER*4 %FILL /1980/
```

For a detailed description of the syntax and use of RECORD statements and structure declarations, see Sections 4.13 and 4.15.

2.2.5.1 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.

Examples

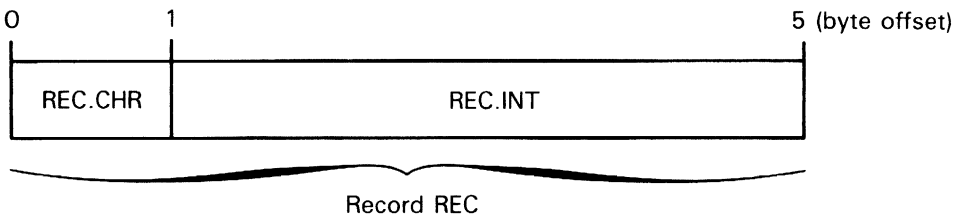
The following examples contain structure declaration blocks, RECORD statements, and diagrams of the resulting records as they are stored in memory.

The first example shows a basic structure declaration block and RECORD statement:

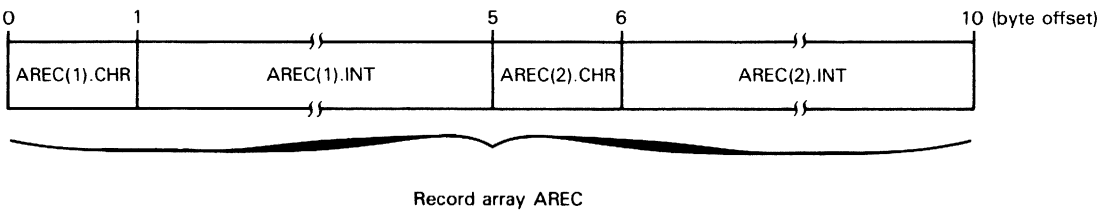
Source Code

```
STRUCTURE /STRA/
  CHARACTER*1 CHR
  INTEGER*4 INT
END STRUCTURE
.
.
.
RECORD /STRA/ REC,AREC(2)
```

Memory Diagram



ZK-1844-84



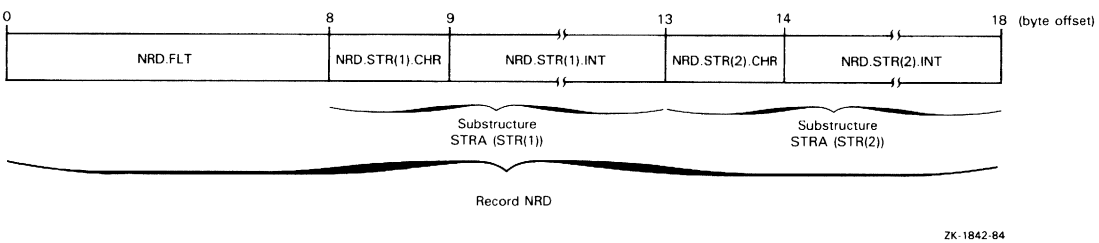
ZK-1843-84

The next example includes a substructure:

Source Code

```
STRUCTURE /STRB/
  REAL*8 FLT
  RECORD /STRA/ STR(2)
END STRUCTURE
.
.
.
RECORD /STRB/ NRD
```

Memory Diagram

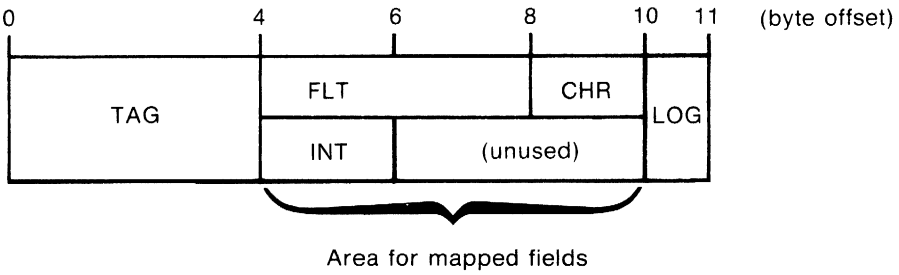


The next example shows how unions cause the storage of the associated mapped fields to be overlaid:

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
.
.
.
```

Memory Diagram



ZK-1845-84

2.2.5.2 References to Record Fields

References to record fields must correspond to the kind of field being referenced. Aggregate field references refer to composite structures (and substructures). Scalar field references refer to singular data items, such as variables.

An operation on a record can involve one or more fields.

Record field references take one of the following forms:

Aggregate Field Reference

```
record-name[.aggregate-field-name]...
```

Scalar Field Reference

```
record-name[.aggregate-field-name]....scalar-field-name
```

record-name
Is the name used in a RECORD statement to identify a record. See Section 4.13 for a description of the RECORD statement.

aggregate-field-name
Is the name of a field that is a substructure (a record or a nested structure declaration) within the record structure identified by the record name.

See Section 4.15 for a description of how fields are specified within structure declarations.

scalar-field-name
Is the name of a typed data item defined within a structure declaration.

Scalar field references are *scalar references* in that they resolve into single, typed data items. Conversely, aggregate field references are *aggregate references* in that they resolve into references to structured data items: records and nested structure declarations. Aggregate field references are the only references that fall into the aggregate reference category. (Section 2.2.6 discusses this terminology.)

Syntax Rules and Behavior

Records and record fields cannot be used in EQUIVALENCE statements. However, you can make fields of record structures equivalent to themselves by using the UNION and MAP statements in a structure declaration block (see Section 4.15.1).

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. However, they cannot be used in COMMON, SAVE, NAMELIST, or EQUIVALENCE statements, or as the control variable in an indexed DO-loop.

Type conversion rules for scalar field references are the same as those for variables and array elements.

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.

You can assign an aggregate field to another aggregate field (record = record) only with records having the same structure. VAX FORTRAN supports no other operations (such as arithmetic or comparison) on aggregate fields.

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.

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 can be used in RECORD statements that are used as dummy arguments.

NOTE

Because periods are used in record references to separate fields, you should not use relational operators (.EQ., .XOR.) logical constants (.TRUE., .FALSE.) and logical expressions (.AND., .NOT., .OR.) as field names in structure declarations.

Examples

The following examples demonstrate record and field references. Consider the following structure declarations (described at length in Section 4.15.1).

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
```

Consider also the following RECORD statement, which creates a variable named NEXT_APP and a 10-element array named APP_LIST. Both the variable and each element of the array take the form of the structure APPOINTMENT.

```
RECORD /APPOINTMENT/ NEXT_APP,APP_LIST(10)
```

Each of the following examples of record and field references are derived from the preceding structure declarations and RECORD statement.

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_MEMO(1)(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
```

2.2.6 Terminology Used to Refer to Data Items

Constants, variables, arrays, array elements, scalar record fields, aggregate fields, character substrings, and expressions can be specified in many places in a VAX FORTRAN program. FORTRAN statements and expressions have individual restrictions governing which of these items can be used in statements and expressions 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 divide 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 into a reference to a single, typed data item (a variable, scalar record field, array element, constant, character substring, or expression).

- *Scalar Memory Reference*—resolves into a data item that can be assigned a value by means of an assignment statement. It is the same as a scalar reference, excluding constants and expressions.
- *Array Name Reference*—resolves into a reference to an array.
- *Aggregate Reference*—resolves into a reference to a structured data item (a record structure or substructure).

Scalar reference, scalar memory reference, array name reference, and aggregate reference are used throughout this manual to indicate where these various categories of data items can be specified.

Examples

Consider the following declarations:

```
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)
```

Each of the following references are derived from the preceding data declarations.

Scalar References

```
INT
INTARY(1)
REC.INTFLD
REC.INTFLDARY(1)
REC.RECFLD.INTFLD
REC.STRUCFLD.CPXFLD
REC.RECFLD.INTFLDARY(1)
REC.RECFLDARY(1).INTFLD
REC.RECFLDARY(1).INTFLDARY(1)
REC.CHARFLD
REC.CHARFLD(5:10)
RECARY(1).CHARFLD(5:10)
RECARY(1).INTFLD
RECARY(1).INTFLDARY(1)
RECARY(1).RECFLD.INTFLD
RECARY(1).STRUCFLD.CPXFLD
RECARY(1).RECFLD.INTFLDARY(1)
RECARY(1).RECFLDARY(1).INTFLD
RECARY(1).RECFLDARY(1).INTFLDARY(1)
```

Scalar Memory References

All references listed in the scalar reference category are also in the category of scalar memory reference because they do not include constants and expressions.

Array Name References

```
INTARY
RECARY
REC.INTFLDARY
REC.RECFLDARY
REC.RECFLD.INTFLDARY
REC.RECFLDARY(1).INTFLDARY
REC.STRUCFLD.CPXFLDARY
RECARY(1).INTFLDARY
RECARY(1).RECFLDARY
RECARY(1).RECFLD.INTFLDARY
RECARY(1).STRUCFLD.CPXFLDARY
RECARY(1).RECFLDARY(1).INTFLDARY
```

Aggregate References

```
REC
RECARY(1)
REC.RECFLD
REC.STRUCFLD
REC.RECFLDARY(1)
RECARY(1).RECFLD
RECARY(1).STRUCFLD
RECARY(1).RECFLDARY(1)
```

2.3 Expressions

An expression is a scalar field reference, function reference, or combination of these references plus operators. Expressions represent singular values. Combinations represent singular values because they resolve into singular values when computations are made on their data items, as specified by the operators.

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.

2.3.1 Arithmetic Expressions

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

An arithmetic element can be any of the following:

- Numeric scalar reference
- Arithmetic expression enclosed in parentheses
- Numeric function reference

The term *numeric* includes logical data, because logical data are 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. Operators and their functions are as follows:

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition or unary plus
–	Subtraction or unary minus

The plus and minus operators are *unary operators* because they can operate on a single operand. Used as unary operators, the plus or minus operators precede a single operand and denote a positive or negative value, respectively. Thus, they preserve or change the arithmetic sign of a value. The exponentiation, multiplication, and division operators are *binary operators* because they operate on a pair of operands.

Variables, array elements, and field references must have defined values before being used in an arithmetic expression.

Valid arithmetic operations must have results that are mathematically defined. For example, dividing by zero or raising a zero-valued base to a zero-valued or negative-valued power is invalid. Raising a negative-valued base to a real power is also invalid.

Arithmetic expressions are evaluated in an order determined by a precedence associated with each operator:

Operator	Precedence
**	First
* and /	Second
+ and –	Third

When operators with equal precedence appear, they can be evaluated in any order as long as the order 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 appear together. However, VAX FORTRAN allows two consecutive operators if the second operator is a plus or minus.

Examples

In the following example, the exponentiation operator is evaluated first because it takes precedence over the multiplication operator:

`A**B*C` is evaluated as `(A**B)*C`.

Ordinarily, the exponentiation operator would be evaluated first in the next example. However, because VAX FORTRAN allows the combination of the exponentiation and minus operators, the multiplication operator is evaluated first. The exponentiation operator must wait until the minus

operator is evaluated. As a result, the multiplication operator is evaluated first, since it takes precedence over the minus operator:

`A**-B*C` is evaluated as `A**(-(B*A))`.

2.3.1.1 Using 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 four examples, the numbers below the operators indicate a possible order of evaluation. Alternative evaluation orders are possible in the first three examples because they contain operators of equal precedence that are not dictated by parentheses. In these cases, the compiler is free to evaluate operators of equal precedence in any order—as long as the result is the same as the result gained by the algebraic left-to-right order of evaluation.

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


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


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


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

As shown in the last two examples, expressions within parentheses are evaluated according to the normal order of precedence, unless you override the order by using parentheses within parentheses.

Nonessential parentheses do not affect expression evaluation, as shown in the following example:

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

However, using 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 (because of the way intermediate results are rounded off).

2.3.1.2 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 and 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. However, 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.
- *Real element to higher-precision element conversions*: 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 approximately to 0.3333331346511840. It is not converted to either 0.3333333000000000 or 0.3333333333333330.
- *Complex operations*: In operations that contain any complex elements, integer elements are converted to the real data type, as previously described. The obtained REAL or REAL*8 element is then designated as the real part of a complex number and the imaginary part is assigned a value of zero. Next, the expression is evaluated using complex arithmetic and the resulting value is a complex data type. Operations involving COMPLEX*8 and REAL*8 elements are done as COMPLEX*16 operations; the REAL*8 element is not rounded.
- *Constants defined by PARAMETER statements*: If a constant was assigned a value by a PARAMETER statement, it may be treated as a lower-order type in an arithmetic expression. This treatment can occur even if the constant was explicitly typed. For example, an INTEGER*4 constant might be treated as ar INTEGER*2 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 $1.0D0 + 0.3333333000000000D0$.

2.3.2 Character Expressions

Character expressions consist of character elements and character operators. The evaluation of a character expression yields a single value with a character data type. Character expressions take the following form:

```
character-element [//character-element]...
```

character-element
Is any one of the following entities:

- Character scalar reference
- Character substring
- Character expression, optionally enclosed in parentheses
- Character function reference

The only character operator is the concatenation operator (//).

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 '.

2.3.3 Relational Expressions

A relational expression consists of two arithmetic expressions or two character expressions separated by a relational operator. A relational operator tests for a relationship between the two expressions. The value of the relational expression is either .TRUE. or .FALSE. depending on whether the stated relationship holds.

VAX FORTRAN supports the following relational operators:

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

Both delimiting periods are required.

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 APPLE and PEACH is greater than the sum of 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'//'ZZZ' .LT. 'CCCCC'
```

This expression states that 'ABZZZ' is less than 'CCCCC'. Because 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 do not need to 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.

2.3.4 Logical Expressions

A logical expression is a single logical element or a combination of logical elements and logical, arithmetic, or relational operators.

Logical elements can be any one of the following:

- Integer or logical scalar reference
- Relational expression
- Integer or logical expression enclosed in parentheses
- Integer or logical function reference

VAX FORTRAN logical operators are any one of the following:

Operator	Example	Meaning
.AND.	`A .AND. B`	Logical conjunction: The expression is true 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 both A and B have the same logical value, whether true or false.
.NOT.	`.NOT. A`	Logical negation: The expression is true if A is false and false if A is true.

Delimiting periods are required. Periods cannot appear consecutively except when the second operator is .NOT. For example, the following logical expression is valid:

```
A+B/(A-1) .AND. .NOT. D+B/(D-1)
```

Data Types that Result from Logical Operations

On logical elements, logical operations produce single logical values (.TRUE. or .FALSE.) with a logical data type.

On integers, logical operations produce single values with an integer data type; they are carried out bit-by-bit on corresponding bits of the internal (binary) representation of the integer elements.

On a combination of integer and logical values, logical operations also produce single values with an integer data type. The operation first converts logical values to integers and then operates as it does with integers.

Logical operations cannot be performed on other data types.

Evaluation of Logical Expressions

Logical expressions are evaluated according to the precedence of their operators. Consider the following expression:

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

This expression is evaluated in the following sequence:

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

The following list identifies all the operators that can appear in a logical expression in the order of their precedence:

Operator	Precedence
**	First (highest)
*, /	Second
+, -, //	Third
Relational Operators	Fourth
.NOT.	Fifth
.AND.	Sixth
.OR.	Seventh
.XOR., .EQV., .NEQV.	Eighth (lowest)

As with arithmetic expressions, you can alter the sequence of evaluation by using parentheses.

When operators have equal precedence, the compiler can evaluate them in any order—as long as the result is the same as the result gained by the algebraic left-to-right order of evaluation (except for exponentiation, which is associated from right to left).

You should not write logical expressions whose results might depend on the evaluation order of subexpressions. The compiler is free to evaluate subexpressions in any order. In the following example, either (A(I)+1.0) or B(I)*2.0 could be evaluated first:

```
(A(I)+1.0) .GT. B(I)*2.0
```

Some subexpressions may not be evaluated if the compiler can determine the result by testing other subexpressions in the logical expression. Consider the following expression:

```
A .AND. (F(X,Y) .GT. 2.0) .AND. B
```

If A is false, and if the compiler evaluates A first, then the compiler can determine that the expression is false and may not call the subprogram F(X,Y).

Chapter 3

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.

VAX FORTRAN supports five assignment statements: arithmetic, logical, character, aggregate, and ASSIGN.

3.1 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. It takes the following form:

```
v = e
```

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*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.

Examples

The following examples demonstrate valid and invalid assignment statements and explain why the invalid ones are not valid:

Valid

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

PI = 3.14159

SUM = SUM + 1.

NEW = RECORD1.FIELD1
```

Invalid	Explanation
`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)`	Expression on the right cannot have a character data type if the entity on the left does not.

Table 3–1 summarizes the data conversion rules for assignment statements.

Table 3–1: Conversion Rules for Assignment Statements

Scalar Memory Reference (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	Same as above	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)

ZK-4812-85

3.2 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.

A logical assignment statement takes the following form:

```
v = e
```

v
Is a logical scalar memory reference.

e
Is a logical, integer, or arithmetic expression.

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

Examples

The following examples demonstrate valid logical assignment statements:

```
PAGEND = .FALSE.

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

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

3.3 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. It takes the following form:

```
v = e
```

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 have a character data type. You cannot assign a numeric value to a character scalar memory reference.

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

The following examples demonstrate valid and invalid character assignment statements and explain why the invalid ones are not valid. (In the examples, all memory references have a character data type.)

Valid

```
FILE = 'PROG2'

REVOL(1) = 'MAR'//'CIA'

LOCA(3:8) = 'PLANT5'

TEXT(I,J+1)(2:N-1) = NAME//X
```

Invalid	Explanation
`'ABC' = CHARS`	Left element must be a character variable, array element, or substring reference.
`CHARS = 25`	Expression on right must have a character data type.
`STRING = 5HBEGIN`	Expression on right must have a character data type—Hollerith constants are numeric, not character.

3.4 Aggregate Assignment Statement

The aggregate assignment statement assigns the value of each field of the aggregate on the right of an equal sign to the corresponding field of the aggregate on the left. Both aggregates must be declared with the same structure.

An aggregate assignment statement takes the following form:

```
v = e
```

v
Is an aggregate reference with the same structure as the aggregate represented by e (see Section 2.2.5.1).

e
Is an aggregate reference with the same structure as the aggregate represented by v (see Section 2.2.5.1).

Example

The following example demonstrates valid aggregate assignments:

```
STRUCTURE /DATE/
   LOGICAL*1 DAY, MONTH
   INTEGER*2 YEAR
END STRUCTURE

RECORD /DATE/ TODAY, THIS_WEEK(7)
STRUCTURE /APPOINTMENT/
   ...
   RECORD /DATE/ APP_DATE
   ...
END STRUCTURE
...
RECORD /APPOINTMENT/ MEETING

DO I = 1,7
   CALL GET_DATE (TODAY)

   THIS_WEEK (I) = TODAY
   THIS_WEEK (I).DAY = TODAY.DAY + 1
END DO
MEETING.APP_DATE = TODAY
```

3.5 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. ASSIGN statements take the following form:

```
ASSIGN s TO v
```

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 as a statement label reference and undefined as an integer variable.

An ASSIGN statement must be executed before the statements in which the assigned variable is used. Additionally, the ASSIGN statement and the statements 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. Once the ASSIGN statement associates a statement label to a variable, arithmetic operations on the variable produce unpredictable run-time behavior; for example:

```
NUMBER = NUMBER + 1
```

To return the variable to the status of an integer variable, you could use the following statement, which dissociates NUMBER from statement 100 and assigns it an integer value of 10:

```
NUMBER = 10
```

Once returning the variable NUMBER to its integer variable status, it can no longer be used in an assigned GO TO statement.

Examples

The following additional examples demonstrate valid ASSIGN statements:

```
ASSIGN 10 TO NSTART

ASSIGN 99999 TO KSTOP

ASSIGN 250 TO ERROR
```

In the last example, ERROR must be previously defined as an integer variable.

Chapter 4

Specification Statements

Specification statements are nonexecutable statements used to allocate and initialize variables, arrays, records, and structures; and to define other characteristics of symbolic names used in a program.

VAX FORTRAN supports the following specification statements:

- BLOCK DATA—initiates a series of statements that establish common blocks and assign initial values to entities in named common blocks (Section 4.1).
- COMMON—defines one or more contiguous areas of storage (Section 4.2).
- DATA—assigns initial values to variables, arrays, and array elements before program execution (Section 4.3).
- Data type declaration statements—explicitly define the data type of specified symbolic names (Section 4.4).
- DIMENSION—defines the number of dimensions in an array and the number of elements in each dimension (Section 4.5).
- EQUIVALENCE—associates two or more entities with the same storage location (Section 4.6).
- EXTERNAL—allows use of user-supplied procedures as arguments to subprograms (Section 4.7).
- IMPLICIT—overrides the implied data type of symbolic names (Section 4.8).
- INTRINSIC—allows use of FORTRAN intrinsic functions as arguments to subprograms (Section 4.9).
- NAMELIST—specifies lists of entities whose values may be read or written in namelist-directed I/O statements; associates the list with specified group-names (Section 4.10).

- PARAMETER—associates a symbolic name with a constant value (Section 4.11).
- PROGRAM—assigns a symbolic name to a main program unit (Section 4.12).
- RECORD—establishes a record with the structure defined by the block of statements in a structure declaration (Section 4.13).
- SAVE—retains values of local variables after a return from a subprogram (Section 4.14).
- Structure declaration block—specifies the structure (form) of a record (Section 4.15).
- VOLATILE—prevents optimizations from being performed on specified variables, arrays, and common blocks (Section 4.16).

4.1 BLOCK DATA Statement

A BLOCK DATA statement initiates a series of specification statements that establish common blocks and assign initial values to the entities in named common blocks.

The BLOCK DATA statement takes the following form:

```
BLOCK DATA [nam]
```

nam
Is a symbolic name.

Syntax Rules and Behavior

A BLOCK DATA statement and its associated specification statements are a special kind of program unit, called a *block data subprogram.* The block data subprogram has the following syntax rules:

- Any of the following specification statements can appear in a block data subprogram:

  COMMON
  DATA
  DIMENSION
  EQUIVALENCE
  IMPLICIT
  PARAMETER
  RECORD

SAVE
Structure declaration
Type declaration statements

- A block data subprogram must not contain any executable statements.
- As with other types of program units, the last statement in a block data subprogram must be an END statement.
- Within a block data subprogram, if a DATA statement initializes any entity in a named common block, the subprogram must have a complete set of specification statements that establishes the common block. However, all of the the entities in the block do not have to be assigned initial values in a DATA statement.
- One block data subprogram can establish and define initial values for more than one common block.
- The name of a block data subprogram can appear in the EXTERNAL statement of a different program unit to force the VMS Linker to search object libraries for the BLOCK DATA program unit at link time.

Example

The following example demonstrates a valid block data subprogram:

```
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
```

4.2 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 takes the following form:

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

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 record names separated by commas.

Syntax Rules and Behavior

Any common block name, blank or otherwise, can appear more than once in one or more COMMON statements in a program unit. The list following each successive appearance of the same common block name is treated as a continuation of the list for the block associated with that name.

You can use array declarators in the COMMON statement to define arrays.

A common block can have the same name as a variable, array, record, structure, or field. However, in a program with one or more program units, a common block cannot have the same name as a function, subroutine, or entry name in the executable program.

When common blocks from different program units have the same name, they share the same storage area when the 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, consider a program unit containing the following statement:

```
COMMON CENTS
```

And consider another program unit containing the following statements:

```
INTEGER*2 MONEY
COMMON MONEY
```

When these program units are combined into an executable program, incorrect results may occur if the 2-byte integer variable MONEY is made to correspond to the lower-addressed two bytes of the real variable CENTS.

Example

In the following example, 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 share the same storage location as HEAT and X in the blank common block. It makes LIMA and R share the same storage location as KILO and Q in BLK1.

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

4.3 DATA Statement

The DATA statement assigns initial values to variables and array elements before program execution. It takes the following form:

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

nlist
Is a list containing any combination of variable names, array names, array element names, character substring names, and implied-DO lists. Elements in the list must be separated by commas.

Subscript expressions and expressions in substring references must be integer expressions containing integer constants and implied-DO variables.

An implied-DO list in a DATA statement takes the following form:

```
(dlist, i=n1,n2[,n3])
```

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 integer constant expressions. The expression can contain implied-DO variables of other implied-DO lists that have this implied-DO list within their ranges.

clist
Is a list of constants separated by commas; clist constants take one of the following forms:

```
c
n * c
```

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.

Syntax Rules and Behavior

The DATA statement assigns the constant values in each clist to the entities in the preceding nlist, from left to right, as they appear in the nlist. The number of constants must equal the number of entities in the nlist.

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

The following list describes the relationship between nlist items and clist items:

- If both the constant value in clist and the entity in nlist have numeric data types, conversion is based on the following rules:
  - If necessary, the constant value is converted 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.

- 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.
- 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.

  When the clist constant and the nlist entity conform to these restrictions, the nlist entity is initialized with the character that has the ASCII code specified by the clist constant. (This behavior lets you initialize a character entity to any 8-bit ASCII code.)
- 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 3–1). 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.

Examples

In the following example, the first DATA statement 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. The last DATA statement uses an implied-DO loop to assign values to the odd numbered elements in the array B.

```
INTEGER A(10), B(10)
CHARACTER BELL, TAB, LF, FF, STARS*6
DATA A,STARS /10*0,'****'/
DATA BELL,TAB,LF,FF /7,9,10,12/
DATA (B(I), I=1,10,2) /5*1/
```

4.4 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 and character type declarations.

Type declaration statements can initialize data in the same way as the DATA statement: by having values, bounded by slashes, listed immediately after the symbolic name of the entity.

Syntax Rules

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.
- Once a symbolic name has been used in a context that implicitly assumes a data type, its assumed type cannot be changed by a type declaration statement.

4.4.1 Numeric Type Declaration Statements

Numeric type declaration statements take the following form:

```
type [*n]v [*n] [/clist/] [,v [*n] [/clist/]]...
```

type
Is any one of the following data type specifiers:

BYTE
LOGICAL
INTEGER
REAL
DOUBLE PRECISION
COMPLEX
DOUBLE COMPLEX

BYTE and LOGICAL*1 are equivalent.

****n***
Is an integer that specifies (in bytes) the length of v. It overrides the length that is implied by the data type.

The value of *n must specify an acceptable length for the type of v, as listed in Table 2.1. BYTE, DOUBLE PRECISION, and DOUBLE COMPLEX data types have one acceptable length; thus, for these data types, the *n specifier is invalid.

If an array declarator is used, the *n specifier must be positioned immediately after the array name.

v
Is the symbolic name of a constant, variable, array, array declarator, statement function, or function subprogram.

clist
Is a list of constants, as in a DATA statement. If v is the symbolic name of a constant, the clist cannot be present. (See Section 4.3.)

Syntax Rules and Behavior

A numeric data type declaration statement can define arrays by including array declarators in the list (see Section 2.2.3.1).

A numeric type declaration statement can assign initial values to variables or arrays if it specifies a list of constants (the clist). The specified constants initialize only the variable or array that immediately precedes them. The clist cannot have more than one element unless it initializes an array. When the clist initializes an array, it must contain a value for every element in the array.

Examples

In the following example, the first statement declares COUNT and SUM as integers, and MATRIX as a two-dimensional array of integers.

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

The next example shows instances where one declaration overrides another. In the first statement, M12*4 and IVEC*4 override INTEGER*2. In the second statement, WX3*4 and WX6*4 override REAL*8. In the third statement, QARRAY is initialized with implicit conversion of the REAL*4 constants to a REAL*16 data type.

```
INTEGER*2 I, J, K, M12*4, Q, IVEC*4(10)
REAL*8 WX1, WXZ, WX3*4, WX5, WX6*4
REAL*16 PI/3.14159QO/, E/2.72QO/, QARRAY(10)/5*0.0,5*1.0/
```

4.4.2 Character Type Declaration Statements

Character type declaration statements take the following form:

```
CHARACTER[*len[,]] v[*len][/clist/][,v[*len][/clist/]]...
```

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.

v

Is the symbolic name of a constant, variable, array, array declarator, statement function, or function subprogram.

clist

Is a list of constants, as in a DATA statement. (See Section 4.3.) If v is the symbolic name of a constant, the clist must not be present.

Syntax Rules and Behavior

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.

When an asterisk length specification *(*) is used for a function name or dummy argument, it assumes the length of the corresponding function reference or actual argument (see Chapter 2). Similarly, when an asterisk length specification is used for the symbolic name of a constant, the name assumes the length of the actual constant it represents. For example, STRING assumes a 9-byte length in the following statements:

```
CHARACTER*(*) STRING
PARAMETER (STRING = 'VALUE IS:')
```

The length specification must range from 1 to 65535. If there is no length specification, a length of 1 is assumed.

Character type declaration statements can define arrays if they include array declarators (see Section 2.2.3.1) in their list. The array declarator goes first if both an array declarator and a length are specified.

A character type declaration statement can assign initial values to variables or arrays if it specifies a list of constants (the clist). The specified constants initialize only the variable or array that immediately precedes them. The clist cannot have more than one element unless it initializes an array. When the clist initializes an array, it must contain a value for every element in the array.

Examples

The following examples demonstrate valid and invalid character type declaration statements:

Valid

The first example specifies an array, NAMES, with one hundred 32-character elements; an array, SOCSEC, with one hundred 9-character elements; and a variable, NAMETY, that is 10 characters long and has an initial value of 'ABCDEFGHIJ'.

```
CHARACTER*32 NAMES(100),SOCSEC(100)*9,NAMETY*10/'ABCDEFGHIJ'/
```

In the next example, the CHARACTER statement specifies two 8-character variables, LAST and FIRST (the PARAMETER statement is described in Section 4.11).

```
PARAMETER (LENGTH=4)
CHARACTER*(4+LENGTH) LAST, FIRST
```

In the next example, the CHARACTER statement specifies an array, LETTER, with twenty-six 1-character elements; and a dummy argument, BUBBLE, that has a passed length defined by the calling program.

```
SUBROUTINE S1(BUBBLE)
CHARACTER LETTER(26), BUBBLE*(*)
```

Invalid

The following CHARACTER statement is invalid because the length specified for BIGCHR is too large and the length specifier for QUEST is not an integer constant expression:

```
CHARACTER*16 BIGCHR*(60000*2), QUEST*(5*INT(A))
```

4.5 DIMENSION Statement

The DIMENSION statement defines the number of dimensions in an array and the number of elements in each dimension. It takes the following form:

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

a(d)
Is an array declarator. (See Section 2.2.3.1.)

Syntax Rules and Behavior

The DIMENSION statement allocates a number of storage elements to each array named in the statement, one storage element to each array element in each dimension. The size 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, the following statement defines ARRAY as having 16 real elements of 4 bytes each and defines MATRIX as having 125 integer elements of 4 bytes each:

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

Although arrays can also be declared in type declaration and COMMON statements, array names can appear in only one array declarator in each program unit. For further information on arrays and the storage of array elements, see Section 2.2.3.

DIMENSION has the same form and effect as the VIRTUAL statement. (The VIRTUAL statement is described in Appendix D.)

Examples

The following examples demonstrate valid DIMENSION statements:

```
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:*)
```

4.6 EQUIVALENCE Statement

The EQUIVALENCE statement partially or totally associates two or more entities in the same program unit with the same storage location. It takes the following form:

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

nlist

Is a list of variables, arrays, array elements, or character substring references, separated by commas. The list must contain at least two of these entities.

Each expression in a subscript or a substring reference must be an integer constant expression. Records, record fields, and dummy arguments cannot be specified in EQUIVALENCE statements.

Syntax Rules and Behavior

The EQUIVALENCE statement causes all of the entities in one parenthesized list to be allocated storage beginning at the same storage location.

Entities having different data types can be associated because 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

In the first example, the 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))
```

In the second example, the 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)
```

4.6.1 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.

Two or more elements of the same array should not be associated with each other in one or more EQUIVALENCE statements. For example, you cannot use an EQUIVALENCE statement to associate the first element of one array with the first element of another array, and then attempt to associate the fourth element of the first array with the seventh element of the other array.

Consider the following valid example:

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

As a result, the entire array TABLE would share part of the storage space allocated to TRIPLE. Table 4–1 shows how these statements align the arrays.

Table 4–1: Equivalence of Array Storage

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

Each of the following statements also aligns the two arrays as shown in Table 4–1:

```
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. Table 4–2 shows how these statements align the arrays.

Each of the following statements also aligns the arrays as shown in Table 4–2:

```
EQUIVALENCE (A, B(4,1))
EQUIVALENCE (B(3,2), A(2,2))
```

Table 4–2: Equivalence of Arrays with Nonunity Lower Bounds

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		

Only in the EQUIVALENCE statement can you identify an array element with a single subscript (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 Table 4–1:

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

4.6.2 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))
```

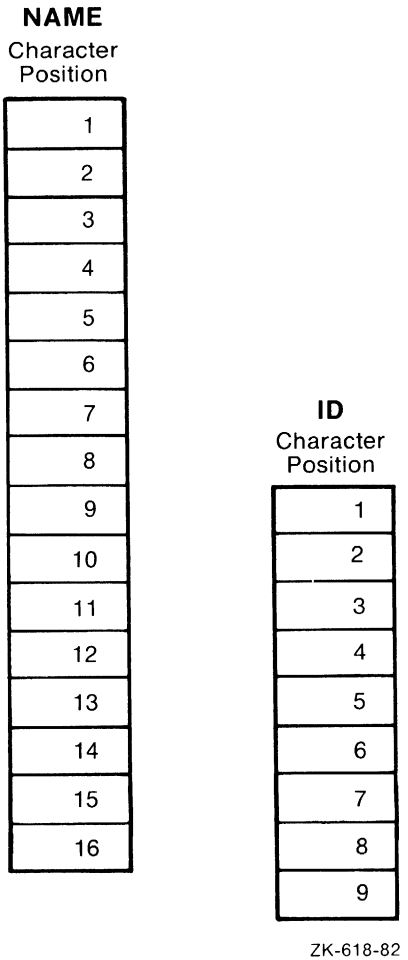
As a result of these statements, the character variables NAME and ID share space as illustrated in Figure 4–1.

The following statement also aligns the arrays as shown in Figure 4–1:

```
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.

Figure 4–1: Equivalence of Substrings

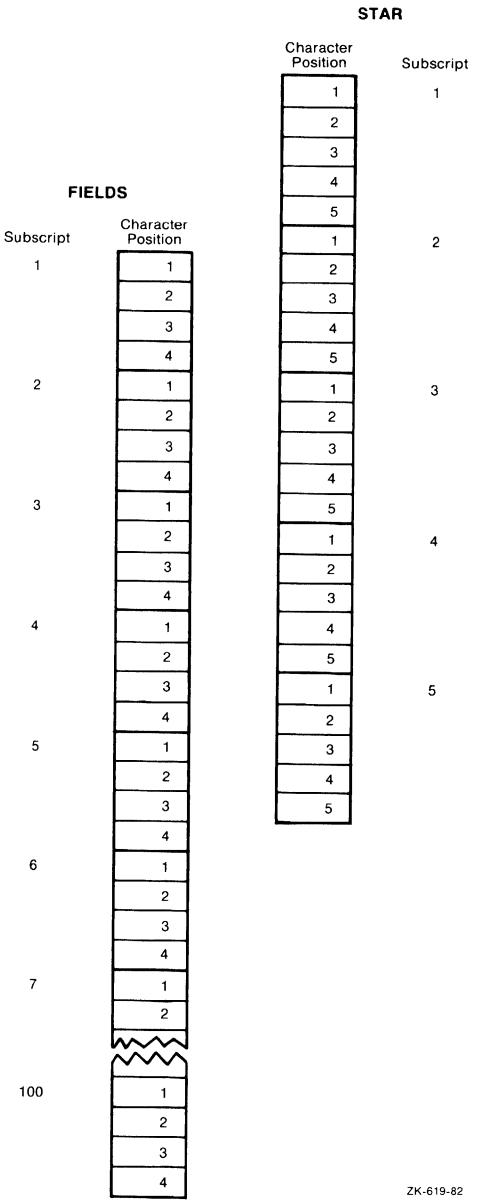


Character elements of arrays can overlap at any character position; for example:

```
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 4–2.

Figure 4–2: Equivalence of Character Arrays



The EQUIVALENCE statement cannot assign the same storage location to two or more substrings that start at different character positions in the same character variable or character array. The EQUIVALENCE statement also cannot assign memory locations in a way that is inconsistent with the normal linear storage of character variables and arrays.

4.6.3 EQUIVALENCE and COMMON Interaction

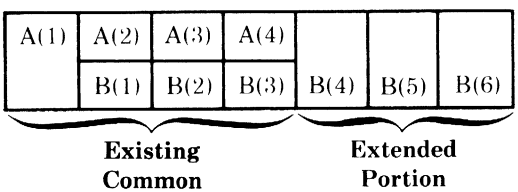
A common block can extend beyond its original boundaries if variables or arrays are associated with entities stored in the common block. However, a common block can only extend beyond its last element; the extended portion cannot precede the first element in the block.

Examples

The following examples demonstrate valid and invalid extensions of the common block:

Valid

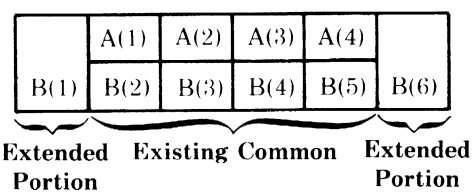
```
DIMENSION A(4), B(6)
COMMON A
EQUIVALENCE (A(2), B(1))
```



ZK-1944-84

Invalid

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



ZK-1945-84

The second example is invalid because the extended portion, B(1), precedes the first element of the common block A.

4.7 EXTERNAL Statement

EXTERNAL statements allow you to use the names of external procedures as arguments to other subprograms.

The subprograms mentioned in the EXTERNAL statement cannot be FORTRAN intrinsic functions; they can only be user-supplied functions, subroutines, or block data subprograms. The INTRINSIC statement discussed in Section 4.9 allows intrinsic function names to be used as arguments.

The EXTERNAL statement takes the following form:

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

v
Is the symbolic name of a user-supplied subprogram or the name of a dummy argument associated with the name of a subprogram.

Syntax Rules and Behavior

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 6.3.1.2).

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 to force the VMS 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.

Used as an argument, a complete function reference represents a value, not a subprogram; for example, FUNC(B) represents a value in the following statement:

```
CALL SUBR (A, FUNC(B), C)
```

A complete function reference cannot be defined in an EXTERNAL statement because there is no data type information.

NOTE

The interpretation of the EXTERNAL statement described here is different from that of earlier versions of FORTRAN produced by DIGITAL. See Appendix A for the earlier interpretation.

See Section 4.9 for an example of EXTERNAL statements.

4.8 IMPLICIT Statement

The IMPLICIT statement overrides implied (default) data typing of symbolic names. (The default data type is INTEGER for symbolic names beginning with the letters I through N, and REAL*4 for symbolic names beginning with any other letter.)

The IMPLICIT statement takes one of the following forms:

```
IMPLICIT typ (a[,a]...)[,typ(a[,a]...)]...
IMPLICIT NONE
```

typ
Is one of the data type specifiers (listed in Section 2.1).

When typ is equal to CHARACTER*len, len specifies the length for character data type. Len is an unsigned integer constant or an integer constant expression enclosed in parentheses; len must be in the range of 1 to 65535.

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.

Syntax Rules and Behavior

The IMPLICIT statement assigns the specified data type to all symbolic names that have no explicit data type and begins with the specified letter or range of letters. It has no effect on the default types of intrinsic functions.

The IMPLICIT NONE statement disables all implicit defaults. When IMPLICIT NONE appears, all symbolic names in a program unit must be explicitly declared. No other IMPLICIT statements can appear in a program unit containing an IMPLICIT NONE statement.

NOTE

By using the /WARNINGS = DECLARATIONS qualifier on the FORTRAN command line, you will be issued warnings when variables are used but not typed—without having to use the IMPLICIT NONE statement, a language extension.

Examples

The following IMPLICIT statements represent the default in the absence of any explicit data type specifications:

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

As above, the following IMPLICIT statements assign the specified data type in the absense of any explicit data type specification:

```
IMPLICIT DOUBLE PRECISION (D)
IMPLICIT COMPLEX (S,Y), LOGICAL*1 (L,A-C)
IMPLICIT CHARACTER*32 (T-V)
IMPLICIT CHARACTER*2 (W)
```

4.9 INTRINSIC Statement

The INTRINSIC statement lets you use names of intrinsic functions as arguments to subprograms. It takes the following form:

```
INTRINSIC v[,v]...
```

v
Is the symbolic name of an intrinsic function.

Syntax Rules and Behavior

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. The subprogram can then use the corresponding dummy argument in a function reference or a CALL statement.

See Appendix D for the names and descriptions of the individual VAX FORTRAN intrinsic functions. For further information on intrinsic functions, see Chapter 6.

Example

In the following example, when TRIG is called with a second argument of SIN or COS, the function reference F(X) references the FORTRAN library functions SIN and COS; but when TRIG is called with a second argument of CTN, F(X) references the user function CTN.

Main Program	Subprogram
.	SUBROUTINE TRIG(X,F,Y)
.	Y = F(X)
EXTERNAL CTN	RETURN
INTRINSIC SIN, COS	END
.	FUNCTION CTN(X)
.	CTN = COS(X)/SIN(X)
CALL TRIG(ANGLE,SIN,SINE)	RETURN
.	END
.	
CALL TRIG(ANGLE,COS,COSINE)	
.	
.	
CALL TRIG(ANGLE,CTN,COTANGENT)	
.	
.	

4.10 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.

NAMELIST statements take the following form:

```
NAMELIST /group-name/ namelist[[,] /group-name/ namelist]...
```

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.

Syntax Rules and Behavior

The namelist associates a group of entities (variables or arrays) with a single group-name, which is used by namelist-directed I/O statements instead of an I/O list. The unique group-name identifies a list whose entities can be modified or transferred.

You cannot include array elements, character substrings, records, and record fields 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.

The namelist entities can have any data type and can be explicitly or implicitly typed.

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.

Example

In the following example, the NAMELIST statement defines two group-names: INPUT, with the entities NAME, GRADE, and DATE; and OUTPUT, with the entities TOTAL and NAME.

```
CHARACTER*30 NAME(25)
NAMELIST /INPUT/ NAME, GRADE, DATE /OUTPUT/ TOTAL, NAME
```

Refer to Sections 7.2.1.3 and 7.3.1.3 for more information on namelist-directed I/O.

4.11 PARAMETER Statement

The PARAMETER statement associates a symbolic name with a constant value. It takes the following form:

```
PARAMETER (p=c[,p=c]...)
```

p
Is a symbolic name.

c
Is a constant, a compile-time constant expression, or the symbolic name of a constant.

NOTE

The form and interpretation of the PARAMETER statement described here are different from those of the PARAMETER statement provided in earlier DIGITAL versions of FORTRAN. 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.

Syntax Rules and Behavior

The data type of a symbolic name associated with 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 (MU has an integer data type by implication):

  ```
  PARAMETER (MU=1.23)
  ```

  If the PARAMETER statement is preceded by an appropriate type declaration or IMPLICIT statement, it could be interpreted as MU=1.23; for example:

  ```
  REAL*8 MU
  PARAMETER (MU=1.23)
  ```

Once a symbolic name is associated with 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 additional rules apply to symbolic 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, although it can appear as either the real or imaginary part of a complex constant.
- A symbolic name can be defined only once within the same program unit.
- A symbolic name defined to be a constant can be used only within the program unit containing the defining PARAMETER statement.

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 with the following characteristics:

- Each operand is either a constant; the symbolic name of a constant; one of the intrinsic 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 with the following characteristics:

- Each operand is either a constant, the symbolic name of a constant, the intrinsic 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 with the following characteristics:

- Each operand is either a constant; the symbolic name of a constant; one of the intrinsic 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 plus, minus, multiplication, division, or exponentiation sign. (The exponentiation operator is evaluated at compile time only if the exponent has an integer data type.)

Example

The following example demonstrates valid FORTRAN-77 PARAMETER statements:

```
REAL*4 PI, PIOV2
REAL*8 DPI, DPIOV2
LOGICAL FLAG
CHARACTER*(*) LONGNAME
PARAMETER (PI=3.1415927, DPI=3.14159265389793238D0)
PARAMETER (PIOV2=PI/2, DPIOV2=DPI/2)
PARAMETER (FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS')
```

4.12 PROGRAM Statement

The PROGRAM statement associates a symbolic name with a main program unit. It takes the following form:

```
PROGRAM nam
```

nam
Is the symbolic name of a source file.

Syntax Rules and Behavior

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 can be preceded only by an OPTIONS statement. Otherwise, it must come first in the main program.

Its symbolic name cannot be the name of any entity within the main program or the name of any subprogram or entry point in the same executable program.

4.13 RECORD Statement

The RECORD statement creates a record having the form specified in a previously declared structure (described in Section 4.15). The effect of using a RECORD statement is comparable to that of an ordinary FORTRAN type declaration, except composite or aggregate data items are declared instead of scalar data items.

RECORD statements take the following form:

```
RECORD /structure-name/record-namelist
     [,/structure-name/record-namelist]
                    .
                    .
                    .
     [,/structure-name/record-namelist]
```

structure-name

Is the name of a previously declared structure. See Section 4.15.1 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.

Syntax Rules and Behavior

You can use record names in the following statements:

COMMON
DIMENSION

Record names 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 (TODAY and YESTERDAY) in separate memory locations, but with the same structure (DATE):

```
RECORD /DATE/ TODAY,YESTERDAY
```

The following RECORD statement creates a record (CURRENT_CHECK) and an array of records (CHECKBOOK) with the structure CHECK:

```
RECORD /CHECK/ CURRENT_CHECK, CHECKBOOK(1000)
```

4.14 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. It takes the following form:

```
SAVE [a[,a]...]
```

a
Is the symbolic name of a common block (preceded and followed by a slash), a variable, or an array.

Syntax Rules and Behavior

A SAVE statement cannot include a blank common block, names of entities in a common block, procedure names, and names of dummy arguments.

Within a program unit, an entity listed in a SAVE statement does not become undefined upon execution of a RETURN or END statement within that program unit.

Even though a common block may be included in a SAVE statement, individual entities within the common block could become undefined (or redefined) in another program unit.

When a SAVE statement does not explicitly contain a list, it is treated as though it contained a list of all allowable items in the program unit that contains it.

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 redundant. 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.

The omission of SAVE statements in necessary instances is not flagged, even when you specify the /STANDARD qualifier on the FORTRAN command line, because the compiler has no way to determine whether such dependencies exist.

4.15 Structure Declaration Block

A structure declaration block is a multistatement declaration that defines the structure (or form) of a record. It contains the following elements:

- STRUCTURE statement — marks the beginning of a structure declaration and defining the name of the structure.
- Declaration body — contains one or more field declarations whose order determines the order of the fields within the structure. Components of the declaration body are individually discussed in this section.

- END STRUCTURE statement — marks the end of a structure declaration.

The Declaration Body

The declaration body can have any of the following components:

- *Typed data declaration statements*: These are ordinary FORTRAN type declarations, as described in Section 4.4. Fields can have any data type and can be dimensioned in the normal way.
- *Substructure declarations*: A field within a structure can be a substructure composed of atomic fields, other substructures, or both. There are two ways to declare substructures:
  - By using RECORD statements that specify names of other, previously declared, structure declarations to be incorporated as substructures; see Section 4.13.
  - By nesting structure declarations (having one or more levels of them contained within a structure declaration).
- *Union declarations*: A union declaration declares that groups of fields logically share a common location within a structure. Each group (one or more fields) is individually declared by a map declaration within the union 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.
- *PARAMETER statements*: PARAMETER statements can appear in a structure declaration block and have the same effect as if they appeared outside the block. (See Section 4.11 for information about PARAMETER statements.)

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.

The following sections describe structure declarations, substructure declarations, and union declarations.

4.15.1 Structure Declaration

Structure declarations define one or more fields within a VAX FORTRAN record. This declaration defines the field names, types of data within fields, and order and alignment of fields within a record.

Unlike type declaration statements, structure declarations do not create variables. Structured variables (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.

A structure declaration takes the following form:

```
STRUCTURE [/structure-name/][field-namelist]
    field-declaration
    [field-declaration]
        .
        .
        .
    [field-declaration]
END STRUCTURE
```

structure-name

Is 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.

Subsequent RECORD statements use the structure name to refer to the structure. A structure name must be unique among structure names. However, structures can share names with variables (scalar or array), record fields, or common blocks. Thus, it is possible to have a variable named X, a structure named X, one or more fields named X, and a common block named X.

Structure declarations can be nested (contain one or more other structure declarations). A structure name is required for the structured declaration at the outermost level of nesting, and optional for the other declarations nested in it. However, if you wish to reference a nested structure in a RECORD statement in your program, it must have a name.

Structure, field, and record 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

Is 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 4.15.2.

field-declaration

Consists of any combination of the following types of declarations:

- *Substructure declaration*: A field within a structure can be a substructure composed of atomic fields, other substructures, or a combination of atomic fields and substructures. See Section 4.15.2 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 4.15.3 for a more complete description of union declarations.
- *Field declaration*: The syntax of a field declaration within a record structure is identical to that of a normal FORTRAN type statement: it includes a type (for example, INTEGER), one or more names of variables or arrays; and optionally, one or more data initialization values.

Syntax Rules and Behavior

The following rules and behavior apply to typed data declarations in record structures:

- %FILL can be specified in place of a field name to leave space in a record for purposes such as alignment. This creates an unnamed field.

  Unnamed fields cannot be initialized. For example, the following statement is invalid and generates an error message:

  ```
  INTEGER*4 %FILL /1980/
  ```

- Initial values can be supplied in 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.
- All field names must be explicitly typed. 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 field 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.

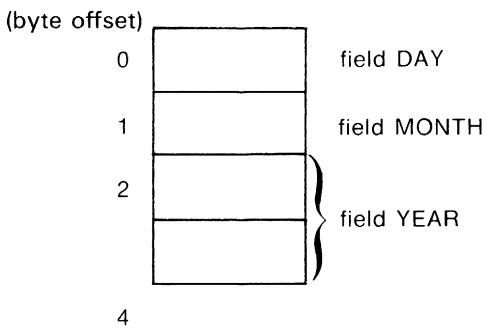
In a structure declaration, each field offset is the sum of the lengths of the previous fields. The length of the structure, therefore, is the sum of the lengths of its fields. The structure is packed; you must explicitly provide any alignment that is needed by including, for example, unnamed fields of the appropriate length.

Examples

In the first example, the declaration defines a structure named DATE. This structure contains three scalar fields: DAY (LOGICAL*1), MONTH (LOGICAL*1), and YEAR (INTEGER*2).

```
STRUCTURE /DATE/
    LOGICAL*1  DAY, MONTH
    INTEGER*2  YEAR
END STRUCTURE
```

The following diagram shows the memory mapping of any record or record array element with the structure DATE.



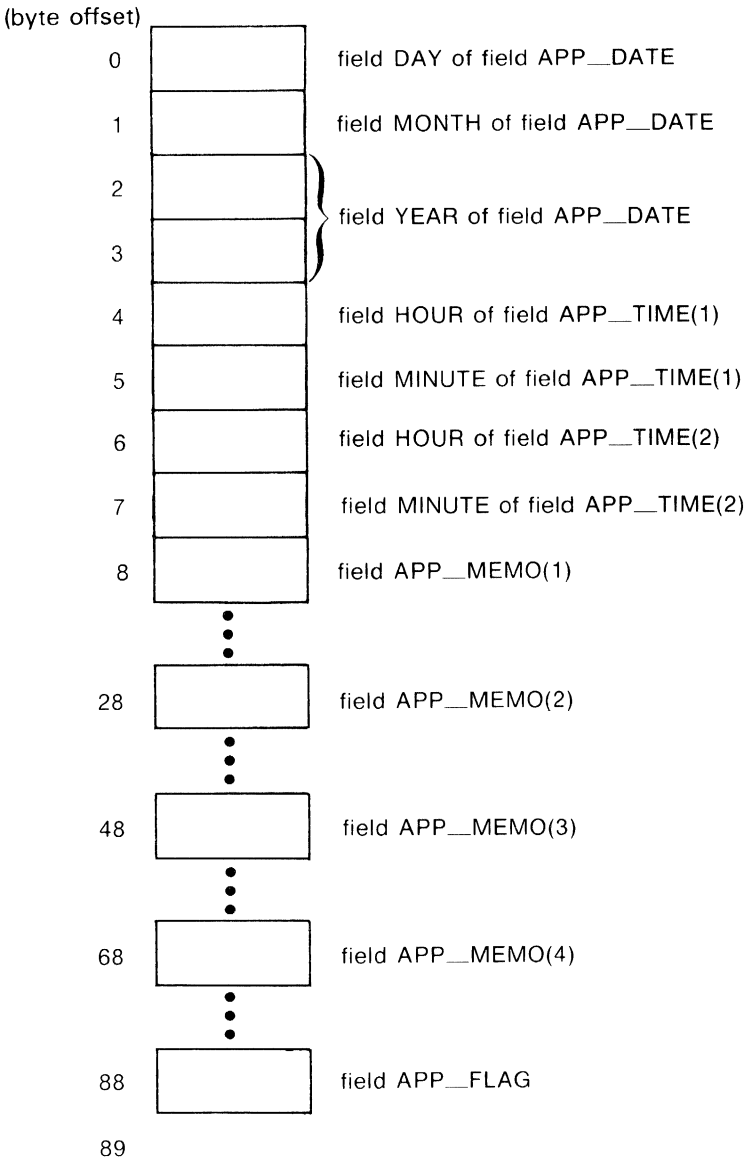
ZK-1849-84

In the second example, the declaration defines a structure named APPOINTMENT. 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 field named APP_FLAG.

```
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
```

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:


(byte offset)
0
field DAY of field APP_DATE
1
field MONTH of field APP_DATE
2
3
field YEAR of field APP_DATE
4
field HOUR of field APP_TIME(1)
5
field MINUTE of field APP_TIME(1)
6
field HOUR of field APP_TIME(2)
7
field MINUTE of field APP_TIME(2)
8
field APP_MEMO(1)
28
field APP_MEMO(2)
48
field APP_MEMO(3)
68
field APP_MEMO(4)
88
field APP_FLAG
89

ZK-1848-84

4.15.2 Substructure Declarations

A field within a structure can itself be a structured item composed of other fields, other structures, or both. You can declare a substructure in two ways:

- By nesting structure declarations within other structure or union declarations (with the limitation that you cannot refer to a structure inside itself at any level of nesting).

  One or more field names must be defined in the STRUCTURE statement for the substructure because all fields in a structure must be named. In this case, 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.

  %FILL can be specified in place of a field name to leave space in a record for purposes such as alignment.

- By using a RECORD statement that specifies another previously defined record structure, thereby including it 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).

4.15.3 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. Each unique field or group of fields is defined by a separate map declaration.

A union declaration takes the following form:

```
UNION
    map-declaration
    map-declaration
   [map-declaration]
     .
     .
     .
   [map-declaration]
END UNION
```

map-declaration

Takes the following form:

```
MAP
    field-declaration
   [field-declaration]
     .
     .
     .
   [field-declaration]
END MAP
```

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 4.15.1 for a more detailed description of what can be specified in field declarations.

Syntax Rules and Behavior

As with normal FORTRAN type declarations, data can be initialized in field declaration statements in union declarations. However, 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 fields 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 are defined within a union at any given point in the execution of a program. However, if you overlay one

variable with another smaller variable, that portion of the initial variable is retained that is not overlaid. Depending on the application, the retained portion of an overlaid variable may or may not contain meaningful data and can 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; union declarations enable you to use one discrete storage location to alternately contain a variety of fields (arrays or variables).

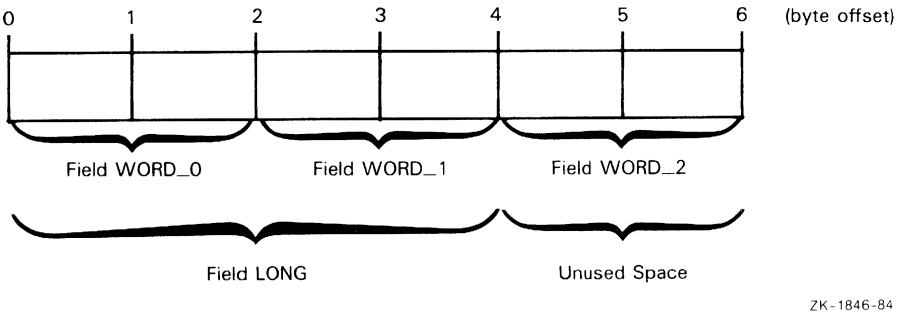
With union declarations, only one map declaration within a union declaration can be associated at any point in time with the storage location that they share. Whenever a field within another map declaration in the same union declaration is referenced in your program, the fields in the prior map declaration become undefined and are succeeded by the fields in the map declaration containing the newly referenced field.

Example

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:



4.16 VOLATILE Statement

The VOLATILE statement specifies that a value is entirely unpredictable, based on information local to the current program unit. It prevents specified variables, arrays, and common blocks from being optimized during compilation.

VOLATILE takes the following form:

```
VOLATILE nlist
```

nlist
Is a list of one or more variable names, named common blocks (preceded and followed by a slash), or array names, separated by commas.

If array names or common block names are used, the entire array or common block becomes volatile, as the following example demonstrates.

Example

In the following statements, the named common block, BLK1, and the variables D and E are volatile. In addition, variables P1 and P4 become volatile because the direct equivalence (in the case of P1) and the indirect equivalence (in the case of P4) causes them to assume the volatile attribute.

```
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)
...
```

See the *VAX FORTRAN User Manual* for information about the optimizations performed by the VAX FORTRAN compiler and the circumstances in which you should use the VOLATILE declaration.

Chapter 5

Control Statements

Statements normally execute in the order in which they are written. However, you can alter the normal order of execution by transferring control to another section of a program unit or a subprogram. Transfer of control can be conditional or unconditional: Conditional transfer occurs only when specified conditions are met at a certain point in a program unit. Unconditional transfer occurs each time a certain point is reached in a program unit.

FORTRAN control statements transfer control to a point within the same program unit or to another program unit. These statements govern iterative processing, suspension of program execution, and program termination.

VAX FORTRAN supports the following control statements:

- CALL—invokes a subroutine subprogram (Section 5.1).
- CONTINUE—transfers control to the next executable statement (Section 5.2).
- DO and DO WHILE—execute a block of statements repetitively (Section 5.3).
- END DO—terminates DO and DO WHILE loops (Section 5.4).
- END—marks the end of a program unit (Section 5.5).
- GO TO—transfers control within a program unit (Section 5.6).
- IF—conditionally transfers control or executes a statement or block of statements (Section 5.7).
- PAUSE—temporarily suspends program execution (Section 5.8).

- RETURN—returns control from a subprogram to the calling program unit (Section 5.9).
- STOP—terminates program execution (Section 5.10).

5.1 CALL Statement

The CALL statement executes a subroutine subprogram or other external procedure. It can specify an argument list for the subroutine. The CALL statement takes the following form:

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

sub
Is the name of a subroutine subprogram or other external procedure, or a *dummy argument* associated with a subroutine subprogram or other external procedure. See Chapter 6 for details on the definition and use of subroutines.

a
Is an *actual argument*. (Section 6.1 describes actual arguments.)

Syntax Rules and Behavior

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

The following examples demonstrate valid CALL statements. The last CALL statement uses statement label identifiers in its argument list. The asterisks 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 6.1.1.5).

```
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)
```

5.2 CONTINUE Statement

The CONTINUE statement transfers control to the next executable statement. It primarily functions as the terminal statement of a labeled DO loop when the loop would otherwise end improperly with either a GO TO, arithmetic IF, or other prohibited control statement.

CONTINUE takes the following form:

```
CONTINUE
```

5.3 DO Statements

DO statements can be either one of two types:

- Indexed (DO)
- Pretested and indefinite (DO WHILE)

5.3.1 Indexed DO Statement

The indexed DO statement controls iterative processing (the statements in its range are repeatedly executed a specified number of times). It takes the following form:

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

s
Is the label of an executable statement. The executable statement must physically follow the DO statement in the same program unit. The label is optional in VAX FORTRAN.

v
Is a variable with an integer or real data type.

e1,e2,e3
Are arithmetic expressions.

Syntax Rules and Behavior

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 5.4.

The terminal statement must not be one of the following statements:

- Unconditional or assigned GO TO
- Arithmetic IF
- Any block IF
- END
- RETURN
- DO

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, or *iteration count,* is given by the arithmetic expression $[(e2 - e1 + e3)/e3]$.

> The notation $[e]$ represents the largest integer whose magnitude does not exceed the magnitude of e and whose sign is the same as the sign of e.

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 executes once.

5.3.1.1 DO Iteration Control

After each iteration of the DO range, the following steps execute:

1. The value of the increment parameter (e3) is algebraically added to the control variable.
2. The iteration count is decremented.
3. The iteration count is evaluated and action taken as follows:
   - 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.
   - 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.

If the control variable has a real data type, the number of iterations of the DO range might not be what you expect because of rounding errors.

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 to the next outermost DO loop in the DO nesting structure (see Section 5.3.1.2). 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 5.3.1.2 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 5.3.1.3 and 5.3.1.4.

Examples

The following examples demonstrate valid and invalid DO iteration control:

Valid

The first statement specifies 25 iterations: K=49 during the final iteration, K=51 after the loop.

```
DO 100 K=1,50,2
```

The next statement specifies 27 iterations: J=–2 during the final iteration, J=–4 after the loop.

```
DO 350 J=50,-2,-2
```

The next statement specifies 5 iterations: IVAR=5 during the final iteration, IVAR=6 after the loop.

```
DO 25 IVAR=1,5
```

The next 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
```

Invalid

The last statement shows how a common typing error can cause errors with DO loops—a decimal point is typed in place of a comma. In effect, this statement assigns 2.10 to the real variable DO40M.

```
DO 40 M=2.10
```

5.3.1.2 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 END DO statement.

Table 5–1 illustrates correctly and incorrectly nested DO loops.

Table 5–1: Nested DO Loops

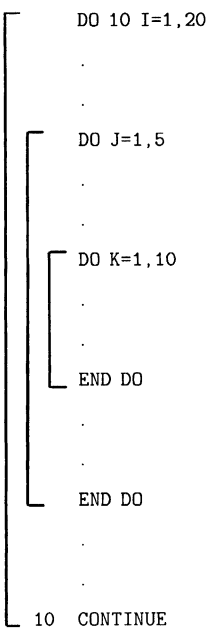
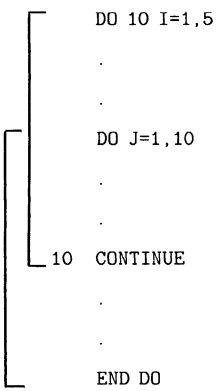
Correctly Nested DO Loops

```
      DO 45 K=1,10
      .
      .
      DO 35 L=2,50,2
      .
      .
35    CONTINUE
      .
      .
      DO 45 M=1,20
      .
      .
45    CONTINUE
```

Incorrectly Nested DO loops

```
      DO 15 K=1,10
      .
      .
      DO 25 L=1,20
      .
      .
15    CONTINUE
      .
      .
      DO 30 M=1,15
      .
      .
25    CONTINUE
      .
      .
30    CONTINUE
```

Table 5–1 (Cont.): Nested DO Loops

Correctly Nested DO Loops	Incorrectly Nested DO loops
	

5.3.1.3 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.

5.3.1.4 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 apply when using a DO loop extended range:

- 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.
- The extended range of a DO statement must not change the control variable of the DO statement.

Figure 5–1 illustrates valid and invalid extended range control transfers.

5.3.2 DO WHILE Statement

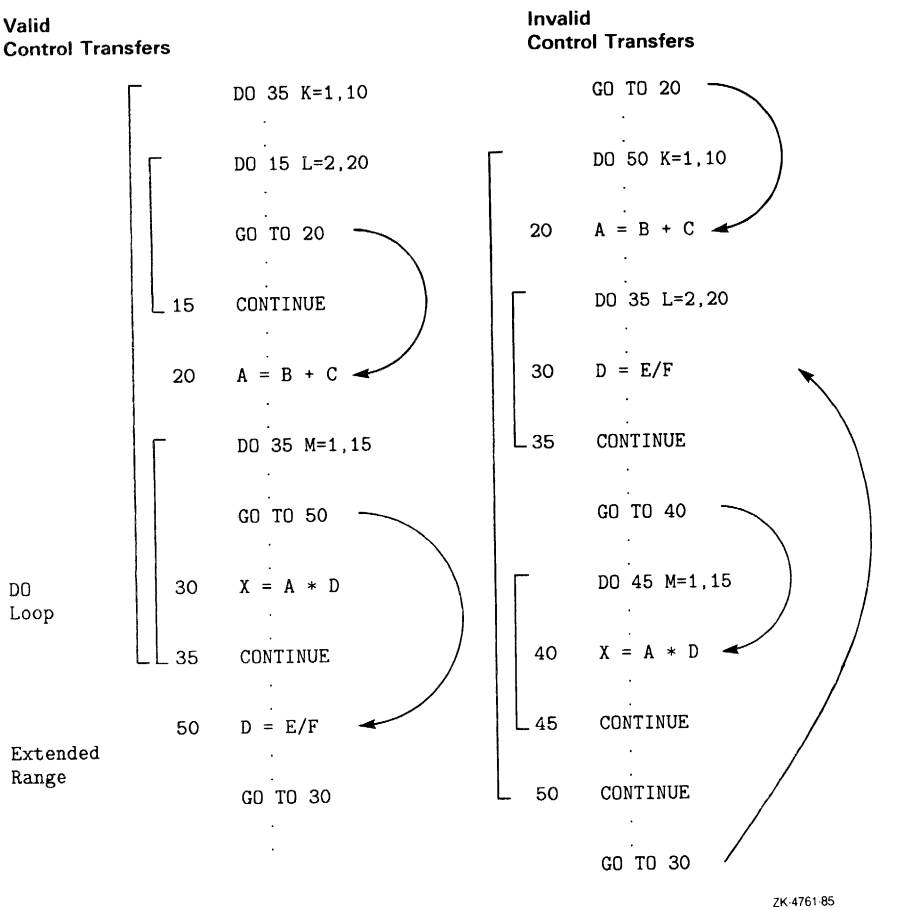
The DO WHILE statement is similar to the DO statement. However, whereas the DO statement executes unconditionally for a fixed number of iterations, the DO WHILE statement executes conditionally for as long as a logical expression contained in it continues to be true. The DO WHILE statement takes the following form:

```
DO [s[,]] WHILE (e)
```

s
Is the label of an executable statement that must physically follow in the same program unit.

e
Is a logical expression.

Figure 5–1: Control Transfers and Extended Range



Syntax Rules and Behavior

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 5.4).

You can transfer control out of a DO WHILE loop but not into a loop from elsewhere in the program.

Example

The following example demonstrates a valid DO WHILE statement:

```
CHARACTER*132 LINE
I = 1
LINE(132:) = 'x'
DO WHILE (LINE(I:I) .EQ. ' ')
    I = I + 1
END DO
```

5.4 END DO Statement

The END DO statement terminates the range of the DO and DO WHILE statements. END DO is a mandatory terminator of DO blocks if the DO or DO WHILE statement defining the blocks does not contain a terminal-statement label. If they do contain a terminal-statement label, END DO is optional and can be used as a labeled terminal statement.

END DO takes the following form:

```
END DO
```

Examples

The following examples demonstrate mandatory and optional END DO statements:

Mandatory

```
DO WHILE (I .GT. J)
    ARRAY(I,J) = 1.0
    I = I - 1
END DO
```

Optional

```
    DO 10 WHILE (I .GT. J)
       ARRAY(I,J) = 1.0
       I = I - 1
10  END DO
```

5.5 END Statement

The END statement marks the end of a program unit and must be the last source line of every program unit. It takes the following 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 only an END in the statement field, it is treated as an END statement even if it is followed by continuation lines.

5.6 GO TO Statements

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

VAX FORTRAN supports three different kinds of GO TO statements:

- Unconditional
- Computed
- Assigned

5.6.1 Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the same statement every time it executes. It takes the following form:

```
GO TO s
```

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

The following examples demonstrate unconditional GO TO statements:

```
GO TO 7734

GO TO 99999
```

5.6.2 Computed GO TO Statement

The computed GO TO statement transfers control to a statement based on the value of an expression within the statement. It takes the following form:

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

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).

Syntax Rules and Behavior

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 statement.

Examples

The following examples demonstrate valid computed GO TO statements:

```
GO TO (12,24,36), INDEX
GO TO (320,330,340,350,360), SITU(J,K) + 1
```

5.6.3 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.

Assigned GO TO statements take the following form:

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

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.

Syntax Rules and Behavior

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.

Both the assigned GO TO statement and its associated ASSIGN statements must exist in the same program unit. Statements that receive control must also be in the same program unit and must be executable.

Examples

The first example is equivalent to `GO TO 200`:

```
ASSIGN 200 TO IGO
GO TO IGO
```

The second example is equivalent to `GO TO 450`:

```
ASSIGN 450 TO IBEG
GO TO IBEG, (300,450,1000,25)
```

5.7 IF Statements

IF statements conditionally transfer control or execute a statement or block of statements. They can be any one of three kinds:

- Arithmetic
- Logical
- Block (IF THEN, ELSE IF THEN, ELSE, END IF)

For each kind, 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.

5.7.1 Arithmetic IF Statement

The arithmetic IF statement conditionally transfers control to one of three statements, based on the current value of an arithmetic expression. It takes the following form:

```
IF (e) s1,s2,s3
```

e
Is an arithmetic expression.

s1,s2,s3
Are labels of executable statements in the same program unit.

Syntax Rules and Behavior

All three labels (s1,s2,s3) are required, but they do not need to 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 of e is:	Control passes to:
Less than 0	Label s1
Equal to 0	Label s2
Greater than 0	Label s3

Examples

The first example 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 (THETA-CHI) 50,50,100
```

The second example 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.

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

5.7.2 Logical IF Statement

The logical IF statement conditionally executes a single FORTRAN statement based on the current value of a logical expression within the logical IF statement. It takes the following form:

```
IF (e) st
```

e
Is a logical expression.

st
Is any complete, executable FORTRAN statement—except any of the block IF statements, DO, END DO, or another logical IF statement.

The logical IF statement first evaluates the logical expression e and then acts as follows:

- If e is true, st executes.
- If e is false, control transfers to the next executable statement after the logical IF; st does not execute.

Examples

The following examples demonstrate valid logical IF statements:

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

IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K) * (-1.5D0)

IF (ENDRUN) CALL EXIT
```

5.7.3 Block IF Statements

Block IF statements conditionally execute blocks (groups) of statements. They can be any one of the following:

- IF THEN
- ELSE IF THEN
- ELSE
- END IF

In block IF constructs, these statements take the following form:

```
IF (e) THEN
  block

ELSE IF (e1) THEN
  block
    .
    .
    .

ELSE
  block

END IF
```

e
Is a logical expression.

block
Is a sequence of zero or more complete FORTRAN statements. This sequence is called a *statement block.*

Syntax Rules and Behavior

Each statement in a block IF construct, 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 values of the logical expressions in the preceding block IF statements.

The functions of individual statements in a block IF construct are as follows:

- IF THEN—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.

NOTE

No additional statement can be placed after the IF THEN statement in a block IF construct. For example, the following statement is invalid in the block IF construct:

```
IF (e) THEN I = J
```

This statement is translated as the logical IF statement `IF (e) THENI = J`.

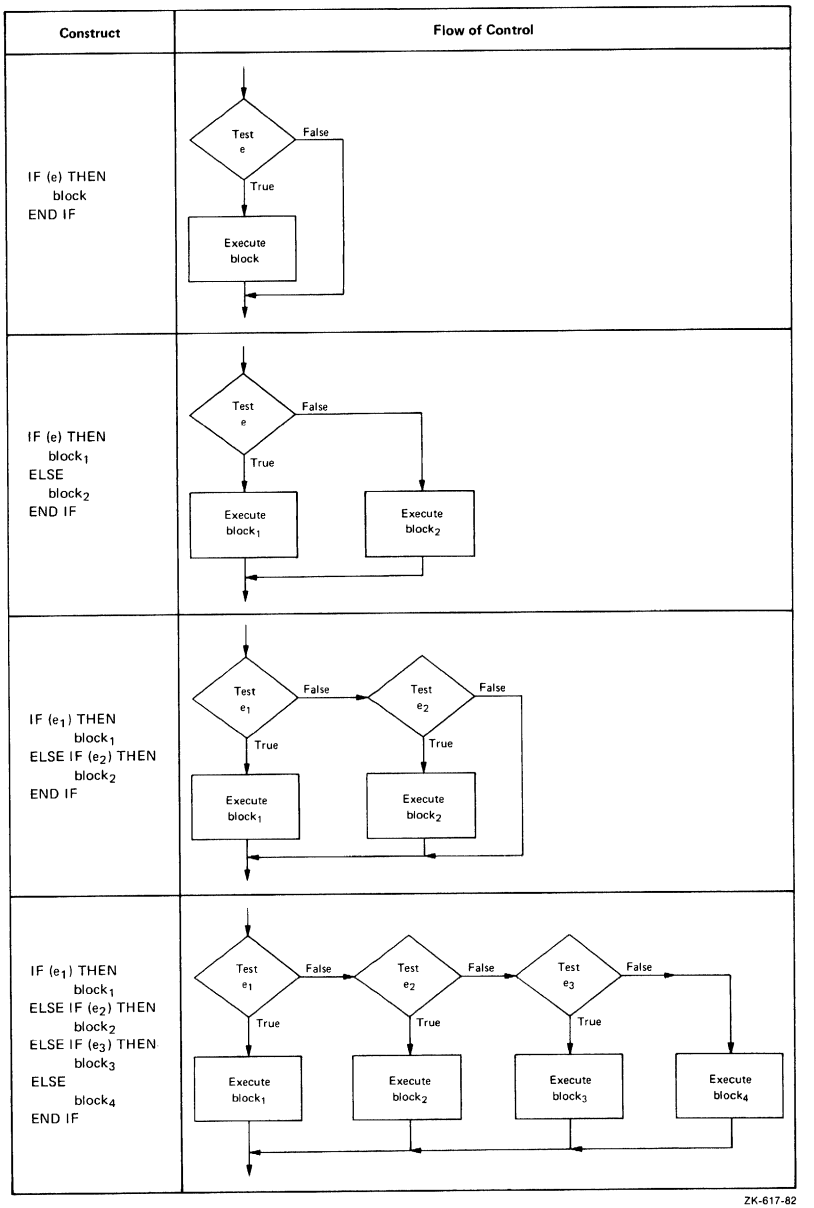
- ELSE IF THEN—is an optional statement that specifies a statement block to be executed if no preceding statement block in the block IF construct has been executed, and if the value of the logical expression in the ELSE IF THEN statement is true. A block IF construct can contain any number of ELSE IF THEN statements.
- ELSE—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.
- END IF—terminates the block IF construct.

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 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.

Figure 5–2 shows the flow of control for four examples of block IF constructs.

Figure 5–2: Examples of Block IF Constructs



5.7.3.1 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. The DO statement and its terminal statement must appear together in a statement block.

5.7.3.2 Block IF Examples

The following examples illustrate four variations of block IF constructs:

Variation 1: The simplest block IF construct consists of the IF THEN and END IF statements. This construct conditionally executes one statement block, which consists of all the statements between the IF THEN and the END IF statements.

Form	Example
IF (e) THEN	`IF (ABS(ADJU) .GE. 1.0E-6) THEN`
block	`TOTERR = TOTERR + ABS(ADJU)`
	`QUEST = ADJU/FNDVAL`
END IF	`END IF`

The IF THEN statement first evaluates the logical expression e. 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.

Variation 2: The second variation contains a block IF construct with an ELSE IF THEN statement. 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.

Form	Example
IF (e1) THEN	`IF (A .GT. B) THEN`
block1	`D = B`
	`F = A - B`
ELSE IF (e2) THEN	`ELSE IF (A .GT. B/2.) THEN`
block2	`D = B/2.`
	`F = A - B/2.`
END IF	`END IF`

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.

Variation 3: The third variation contains a block IF construct with an ELSE statement. 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.

Form	Example
IF (e) THEN	`IF (NAME .LT. 'N') THEN`
block1	`IFRONT = IFRONT + 1`
	`FRLET(IFRONT) = NAME(1:2)`
ELSE	`ELSE`
block2	`IBACK = IBACK + 1`
END IF	`END IF`

Variation 4: The fourth variation contains a block IF construct with several ELSE IF THEN statements and an ELSE statement.

There are four statement blocks in this example. Each consists of all the statements between the block IF statements that follow:

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.

Form	Example
IF (e1) THEN	`IF (A .GT. B) THEN`
block1	`D = B`
	`F = A - B`
ELSE IF (e2) THEN	`ELSE IF (A .GT. C) THEN`
block2	`D = C`
	`F = A - C`
ELSE IF (e3) THEN	`ELSE IF (A .GT. Z) THEN`
block3	`D = Z`
	`F = A - Z`
ELSE	`ELSE`
block4	`D = 0.0`
	`F = A`
END IF	`END IF`

5.7.3.3 Nested Block IF Constructs

You can include a block IF construct in the statement block of another block IF construct—but the nested block IF construct must be completely contained within a statement block. It cannot overlap statement blocks.

The following example contains a nested block IF construct:

	Form	Example
	IF (e1) THEN	IF (A .LT. 100) THEN
block 1		INRAN = INRAN + 1
	IF (e2) THEN	IF (ABS(A-AVG) .LE. 5.) THEN
	block1a	INAVG = INAVG + 1
	ELSE	ELSE
	block1b	OUTAVG = OUTAVG + 1
	END IF	END IF
	ELSE	ELSE
block 2		OUTRAN = OUTRAN + 1
	END IF	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.

5.8 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. It takes the following form:

```
PAUSE [disp]
```

disp
Is a character constant or a string of decimal numbers (one to five digits).

Syntax Rules and Behavior

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 DCL commands:

- CONTINUE—resume execution at the next executable statement.
- EXIT—terminate execution.
- DEBUG—resume execution under control of the VMS Debugger.

5.9 RETURN Statement

The RETURN statement transfers control from a subprogram to the program that called the subprogram. You can use RETURN only in a subprogram unit.

The RETURN statement takes the following form:

```
RETURN [i]
```

i

Is an optional integer constant or expression (such as 2 or I+J) that is converted to an integer value if necessary.

Syntax Rules and Behavior

The optional argument, i, indicates an alternate return from the subprogram. It can be specified only in subroutine subprograms, not in function subprograms. The value of i specifies that the ith alternate return in the actual argument list is to be taken (see the second example that follows in this section).

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 6). 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 ith alternate return in the CALL argument list.

Examples

In the first 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
```

The second 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
```

Depending on the computed value of Z, one of the following returns occurs:

- If Z is less than zero, a normal return occurs and the calling program continues at the first executable statement following CALL CHECK.
- If Z equals zero, the first alternate return (RETURN 1) occurs.
- If Z is greater than zero, the second alternate return (RETURN 2) occurs.

Control returns to the statement specified as the first or second alternate return argument in the CALL statement argument list; for example, the CALL statement might take the following form:

```
CALL CHECK(A,B,*10,*20,C)
```

In this case, RETURN 1 transfers control to statement label 10 and RETURN 2 transfers control to statement label 20.

If a subroutine includes an alternate return specifying a value less than 1 or greater than the number of alternate return arguments, control returns to the next executable statement after the CALL statement (alternate returns are ignored). Thus, you should ensure that the value of i is within the range of alternate return arguments.

5.10 STOP Statement

The STOP statement terminates program execution. It takes the following form:

```
STOP [disp]
```

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

The following examples demonstrate valid STOP statements:

```
STOP 98

STOP 'END OF RUN'
```

Chapter 6

Subprograms — Subroutines and Functions

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. Normally, the program unit 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.

Subprograms are either written by the user or supplied as part of the VAX FORTRAN library.

User-written 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*—Program units, also called functions, that contain 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*—Program units, also called subroutines, that contain 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 as part of the FORTRAN library are called *intrinsic functions* and include the following categories:

- Mathematical
- Character-handling
- Lexical-comparison

6.1 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 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.

Section 6.1.1 describes the general techniques used to pass arguments between FORTRAN programs. Section 6.1.2 describes how to use built-in functions, supplied by VAX FORTRAN, to pass arguments between VAX FORTRAN subprograms and subprograms written in other languages.

6.1.1 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 6.1.1.1) 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. If the character dummy argument's length is specified as *(*), it uses the length of the associated actual argument. (This is known as a passed-length character argument. See Section 6.1.1.3.)

The following sections discuss several kinds of arguments:

- Adjustable array
- Assumed-size array
- Passed-length character
- Character and Hollerith constant
- Alternate return

6.1.1.1 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 2.2.3.1) 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.

Argument association is not retained between one reference to a subprogram and the next reference to that subprogram.

The size of the adjustable array must be less than or equal to the size of the array that is its corresponding actual argument.

Examples

The following examples demonstrate valid and invalid adjustable arrays:

Valid

In the first example, the function computes the sum of the elements of a two-dimensional array. Notice how the dummy arguments M and N 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)
```

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.

The subroutine subprogram in the next example includes statements of a calling program unit. It illustrates how argument association is not retained between one reference to a subprogram and the next reference.

```
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
```

The following program unit calls the subroutine subprogram from the previous example. This 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.

```
DIMENSION B(10)
   .
   .
   .
CALL S(B,2,3.0)
   .
   .
   .
CALL S1(5,B,3,2)
```

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. In the next 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.

```
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
```

Invalid

The following example is invalid—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.

```
SUBROUTINE SUB1(A,X)
DIMENSION A(X)
INTEGER X
```

6.1.1.2 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 noncharacter 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 $INT(n + 1 - b)/y$, where y is the length of an element of the dummy array.

Because the actual size of an assumed-size array is unknown, an assumed-size array name cannot be used as any of the following items in an I/O statement:

- Array name in the I/O list
- Unit identifier for an internal file
- Run-time format specifier

RECORD statements not contained within structure declaration blocks can also declare adjustable arrays.

6.1.1.3 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.

Examples

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.

(The processor-defined function LEN determines the length of the argument. See Section 6.3.2.1 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))
```

6.1.1.4 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.

Example

The following example shows character and Hollerith constants being used as actual arguments. 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

```
SUBROUTINE S(CHARSUB,HOLLSUB,A,B)
EXTERNAL CHARSUB,HOLLSUB
    .
    .
    .

CALL CHARSUB(A,'STRING')
CALL HOLLSUB(B,6HSTRING)
```

6.1.1.5 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 statement must include alternate return arguments in the corresponding positions. These arguments take either one of the following forms:

```
*label
&label
```

Either an asterisk or an ampersand can 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 unit that issues the CALL statement.

6.1.2 Built-In Functions

Built-in functions perform utility operations that are useful in communicating with subprograms written in languages other than FORTRAN. VAX FORTRAN provides the following built-in functions:

- Argument list
- %LOC

6.1.2.1 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 following built-in functions in the argument list of a CALL statement or function reference: %VAL, %REF, %DESCR.

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—and in no other context.

The three argument list built-in functions have the following effects:

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.

a is an actual argument.

See the *VAX FORTRAN User Manual* for more information on argument-passing mechanisms.

Table 6–1 lists the FORTRAN argument-passing defaults and the allowed uses of %VAL, %REF, and %DESCR.

Table 6–1: Argument List Built-In Functions and Defaults

		Allowed Functions		
Actual Argument Data Type	**Default**	**%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
Hollerith	REF	No	No	No
Aggregate	REF	No	Yes	No
Array Name				
Numeric	REF	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.

Table 6–1 (Cont.): Argument List Built-In Functions and Defaults

Actual Argument Data Type	Default	Allowed Functions %VAL	%REF	%DESCR
Character	DESCR	No	Yes	Yes
Aggregate	REF	No	Yes	No
Procedure Name				
Numeric	REF	No	Yes	Yes
Character	DESCR	No	Yes	Yes

6.1.2.2 %LOC Built-In Function

The %LOC built-in function computes the internal address of a storage element. It takes the following form:

```
%LOC(arg)
```

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 Manual* for more information on the %LOC built-in function.

6.2 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. A single subprogram can perform a computing procedure in several places in your program, and thus avoid duplicating a series of operations or statements in each place.

There are three types of subprograms. Table 6–2 lists each type of subprogram, the statements needed to define the subprogram, and the method of transferring control to it.

Table 6–2: Types of User-Written Subprograms

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 refer to itself, either directly or indirectly.

6.2.1 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.

Statement function definitions take the following form:

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

fun
Is the symbolic name of the statement function.

p
Is a dummy argument.

e
Is an arithmetic, logical, or character expression that defines the computation to be preformed.

Statement function references take the following form:

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

f
Is the symbolic name of the statement function.

p
Is an actual argument.

Syntax Rules

The following rules apply to statement function definitions and references:

- 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. The names of the dummy arguments can represent other entities elsewhere in the program unit.

  Except for the data type, declarative information associated with an entity is not associated with dummy arguments in the statement function: 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.

- The name of the statement function cannot 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, it must be previously defined 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. The reference cannot appear on the left side of an assignment statement.

Examples

The following statement function definitions are valid:

```
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 statement function definition is invalid because it contains a constant, which cannot be used as a dummy argument:

```
AVG(A,B,C,3.) = (A+B+C)/3.
```

Consider the following definition:

```
AVG(A,B,C) = (A+B+C)/3.
```

Based on the previous statement, the following references are valid:

```
      .
      .
      .
GRADE = AVG(TEST1,TEST2,XLAB)
IF (AVG(P,D,Q) .LT. AVG(X,Y,Z)) GO TO 300
```

The following reference is invalid because the data type of the third argument does not agree with the dummy argument:

```
FINAL = AVG(TEST3,TEST4,LAB2)
```

6.2.2 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. Function references transfer control to a function subprogram; RETURN or END statements 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.

6.2.2.1 Logical and Numeric Functions

The FUNCTION statement takes the following form:

```
[typ] FUNCTION nam[*m] [([p[,p]...])]
```

typ
Is one of the logical or numeric data type specifiers. See Section 4.4.1 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.

6.2.2.2 Character Functions

The CHARACTER FUNCTION statement takes the following form:

```
CHARACTER[*n] FUNCTION nam[*n] [([p[,p]...])]
```

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.

6.2.2.3 Function Reference

A function reference that transfers control to a function subprogram takes the following form:

```
nam([p[,p]...])
```

nam
Is the symbolic name of the function.

p
Is an actual argument.

When control transfers to a function subprogram, the values of any actual arguments in the function reference are associated with any dummy arguments in the FUNCTION statement. The statements in the subprogram are then executed. 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; it can also 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 6.2.4).

Examples

Consider 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
```

To obtain the root of the function, the previous example uses the Newton-Raphson iteration method:

$$F(X) = cosh(X) + cos(X) - A = 0$$

The value of A is passed as an argument. The iteration formula for this root is as follows:

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

This formula 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 6.3).

The next 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
```

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')
```

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')
```

6.2.3 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. The CALL statement transfers control to a subroutine subprogram; a RETURN or END statement returns control to the calling program unit.

SUBROUTINE statements take the following form:

```
SUBROUTINE sub [([p[,p]...])]
```

sub
Is the symbolic name of the subroutine.

p
Is a dummy argument. An asterisk in the argument list specifies a dummy argument as an alternate return argument.

When control transfers to the subroutine, the values of any actual arguments in the CALL statement are associated with any corresponding dummy arguments in the SUBROUTINE statement. The statements in the subprogram are then executed. (Section 5.1 describes the CALL statement.)

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 6.2.4).

Examples

The first 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 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 (6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,6,6,5), 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 next example uses alternate return specifiers to determine where control transfers 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 executes depends on the value of Z, as computed in the subroutine:

- If Z is less than zero, the normal return executes.
- 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	Subroutine
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 CHECK(X,Y,*,*,Q) . . . 50 IF (Z) 60,70,80 60 RETURN 70 RETURN 1 80 RETURN 2 END

6.2.4 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 takes the following form:

```
ENTRY nam[([p[,p]...])]
```

nam
Is the symbolic name of an entry point.

p
Is a dummy argument.

Syntax Rules

The following rules apply to ENTRY statements and names:

- CALL statements should be used to refer to entry names within subroutine subprograms.
- Function references should be used to refer to entry names within function subprograms.
- An entry name within a function subprogram can appear in a type declaration statement.

- An EXTERNAL statement can specify an entry name and use it as an actual argument—but not as a dummy argument.
- Entry names cannot appear in executable statements that physically precede their appearance in an ENTRY statement.
- Alternate return arguments can be included in ENTRY statements if they have asterisks in the dummy argument list. ENTRY statements that specify alternate return arguments can be used only in subroutine subprograms.
- Dummy arguments can be used in ENTRY statements even if they differ in order, number, type, and name from the dummy arguments used 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.
- Dummy arguments can be referred to only in executable statements that follow the first SUBROUTINE, FUNCTION, or ENTRY statement in which the dummy argument is specified. If a dummy argument is not currently associated with an actual argument, the dummy argument is undefined and cannot be referenced—arguments do not retain their association from one reference of a subprogram to another.
- ENTRY statements cannot appear within a block IF construct or a DO loop.

6.2.4.1 ENTRY Statements 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 with the same data type. All associated names with different data types become undefined.

The function and entry names do not need to have 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 an 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 has a character data type, all entry names must have a character data type and the same length specification as the function. The specified length 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 6.1.1.3 and the *VAX FORTRAN User Manual*).

Example

The following example 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
```

6.2.4.2 ENTRY Statements in Subroutine Subprograms

To refer to an entry point name in a subroutine, you should issue a CALL statement that includes the entry point name defined in the ENTRY statement. In the following 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.

Main Program	Subroutine
`CALL SUBA(A,B,C)`	`SUBROUTINE SUB(X,Y,Z)`
`.`	`.`
`.`	`.`
`.`	`.`
	`ENTRY SUBA(Q,R,S)`

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, the RETURN 1 statement transfers control to statement label 100 and the RETURN 2 statement transfers control to statement label 200 in the calling program.

6.3 FORTRAN Intrinsic Functions

FORTRAN intrinsic functions perform frequently used mathematical computations. They are supplied in the VAX FORTRAN library.

References to FORTRAN intrinsic functions use the same form as references to user-defined functions. For example, the following intrinsic function reference is valid:

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

This reference to the intrinsic function ABS causes the absolute value of X-1 to be multiplied by the constant 3.14159; the result of that calculation is assigned to the variable R.

Two methods of referencing intrinsic functions are described in the following sections.

A listing of the intrinsic functions, their data types, and the data type of their actual arguments is located in Appendix D. Further information describing intrinsic function algorithms is located in the *VMS Run-Time Library Routines Volume*.

6.3.1 Intrinsic Function References

FORTRAN library function names are called intrinsic function names. Normally, a name in the table of intrinsic function names (Table D–3) 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 4.7).

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.

Intrinsic and user-defined functions cannot have the same name if they appear in the same program unit.

6.3.1.1 Generic References to Intrinsic Functions

Many of the intrinsic functions supplied with VAX FORTRAN have generic names, 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. The compiler then 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.

However, 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 source code listing.

Function selection occurs independently for each generic reference. Thus, you can use a generic reference repeatedly in the same program unit to access different intrinsic functions.

Table 6–3 lists generic intrinsic function names. However, you cannot use the names in this table to generically select intrinsic functions if you use them in any of the following ways:

- As the name of a statement function
- As a dummy argument name, a common block name, or a variable or array name

Using the generic name of an intrinsic function in an INTRINSIC statement (see Section 4.9) does not affect function references. However, when you use a generic function name in an actual argument list as the name of a function to be passed, 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 handling specific intrinsic function names described in Section 6.3.

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 6–3: Summary of Generic Intrinsic Function Names

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

6.3.1.2 Using Intrinsic Function Names

Example 6–1 shows the different ways to use intrinsic function names. In this annotated example, a single executable program uses the name SIN in four distinct ways:

- As the name of a statement function
- As the generic name of an intrinsic function
- As the specific name of an 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.

Example 6–1: Using Multiple Function Names

```
C     Compare ways of computing sine.

      PROGRAM SINES
      REAL*8 X, PI
      PARAMETER (PI=3.14159265358979323 8D0)
      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)
```

Example 6–1 Cont'd. on next page

Example 6–1 (Cont.): Using Multiple Function Names

```
C     INTRINSIC FUNCTION SINE AS ACTUAL ARGUMENT ❺

      CALL SUB(REAL(Y),SIN)
      END

      SUBROUTINE SUB(A,S)

C     Declare SIN as name of user function ❻

      EXTERNAL SIN

C     Declare SIN as type REAL*8 ❼

      REAL*8 SIN
      COMMON V(3)

C     Evaluate intrinsic function SIN ❽

      V(2) = S(A)

C     Evaluate user-defined SIN function ❾

      V(3) = SIN(A)
      END

C     Define the user SIN function ❿

      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
```

Notes:

❶ A statement function named SIN is defined in terms of the generic function name COS. Because the argument of COS is double precision, the double-precision cosine function is evaluated. The statement function SIN is itself single precision.

❷ The statement function SIN is called.

❸ The name SIN is declared intrinsic so that the single-precision intrinsic sine function can be passed as an actual argument at ❺.

❹ The generic function name SIN is used to refer to the double-precision sine function.

❺ The single-precision intrinsic sine function is used as an actual argument.

❻ The name SIN is declared a user-defined function name.

❼ The type of SIN is declared double precision.

❽ The single-precision sine function passed at ❺ is evaluated.

❾ The user-defined SIN function is evaluated.

❿ The user-defined SIN function is defined as a simple Taylor series using a user-defined function FACTOR to compute the factorial function.

6.3.2 Character and Lexical Comparison Library Functions

Character library functions take character arguments and return integer, ASCII, or character values; lexical comparison library functions take character arguments and return logical values.

6.3.2.1 Character Functions

FORTRAN provides four character functions: LEN, INDEX, ICHAR, and CHAR.

LEN Function

The LEN function returns the length of a character expression. It takes the following form:

```
LEN(c)
```

c
Is a character expression. The value returned indicates how many bytes there are in the expression.

The following example uses the LEN function:

```
      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

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. INDEX takes the following form:

```
INDEX(c1,c2)
```

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.

The following example uses the INDEX function:

```
      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
```

ICHAR Function

The ICHAR function converts a character expression to its equivalent ASCII code and returns the ASCII value. It takes the following form:

```
ICHAR (c)
```

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. It takes the following form:

```
CHAR (i)
```

i
Is an integer expression.

6.3.2.2 Lexical Comparison Functions

FORTRAN provides four lexical comparison functions:

- 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)

Lexical functions take the following form:

```
func(c,c)
```

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.

Example

The following example illustrates a valid lexical comparison function:

```
CHARACTER*10 CH2
IF (LGT(CH2,'SMITH')) STOP
```

The IF statement in the example is equivalent to the following IF statement:

```
IF (CH2 .GT. 'SMITH') STOP
```

Chapter 7

I/O Statements

The following VAX FORTRAN I/O (input/output) statements initiate data transfer operations:

- READ
- WRITE
- REWRITE
- ACCEPT
- TYPE and PRINT

Following a discussion of the basic components of complete I/O statements, this chapter discusses individual I/O statements. For information on other statements that influence data transfer operations, but do not directly initiate them, see Chapters 8 and 9.

7.1 Components of I/O Statements

I/O statements have three basic components: the *statement keyword*, the *control list*, and the *I/O list*. The I/O statement keywords specifically control either input or output operations, as follows:

Input Operations	Output Operations
READ	WRITE
ACCEPT	REWRITE
	TYPE
	PRINT

The control list and I/O list are described separately in the next two sections.

7.1.1 Control List

The control list is composed of one or more parameters that specify the following:

- Logical unit to be acted upon
- Internal file to be acted upon
- Whether formatting is to be used for data editing, and if it is, the format specification
- NAMELIST group-name specification
- Number of a direct access record to be accessed
- Key and key-of-reference of a keyed access record to be accessed
- Name of the variable that contains the completion status of an I/O operation
- Label of the statement that receives control if an error or end-of-file condition occurs

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 takes the following form:

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

p
Is a specifier taking the following form:

[keyword =] value
Is one of several possible keywords and values that are described in the following sections.

7.1.1.1 Syntax Rules for Control-List Parameters

In VAX FORTRAN I/O statements, control-list parameters have three different forms (keyword, nonkeyword, and mixed) with the following syntax rules:

- *Keyword Form*: When the control list parameter is specified with a keyword and equal sign, control-list parameters can appear in any order in the control list.
- *Nonkeyword Form*: When the control-list parameter is specified without a keyword or equal sign, either the logical unit specifier or the internal file specifier must occupy the first (leftmost) 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.

  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 thus occupy the first position in the control list).
- *Mixed Forms*: When keyword and nonkeyword forms are mixed in the same I/O statement, nonkeyword rules apply to the control list.

7.1.1.2 Logical Unit Specifier

The logical unit specifier identifies the logical unit to be accessed. It takes either one of the following forms:

```
[UNIT=]u
[UNIT=]*
```

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 use.

****
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 9.1)
- Implicitly by the system (see the *VAX FORTRAN User Manual* for more information on implicit logical assignments).

7.1.1.3 Internal File Specifier

The internal file specifier identifies the internal file to be used. It takes the following form:

```
[UNIT=]cv
```

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 the *VAX FORTRAN User Manual* for more information on internal files.

7.1.1.4 Format Specifiers

The format specifier stipulates either explicit or list-directed formatting. In the case of explicit formatting, it also identifies the parameter that controls formatting. The format specifier takes either one of the following forms:

```
[FMT=]f
[FMT=]*
```

f
Is the statement label of a FORMAT statement; an integer variable that has been assigned a FORMAT statement label with an ASSIGN statement; the name of an array or array element containing a run-time format; or a 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 8 describes FORMAT statements. Section 8.8 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 7.2.1.2 and 7.3.1.2).

7.1.1.5 Namelist Specifier

The namelist specifier stipulates namelist-directed I/O. It identifies the group-name of the list of entities that may be modified on input or written on output.

The namelist specifier takes the following form:

```
[NML=]group-name
```

group-name
Is the name of a list previously defined in a NAMELIST statement.

The keyword NML is optional only if the following conditions are true:

- The first parameter is a logical unit specifier without an optional UNIT keyword.
- The namelist specifier is the second parameter in the control list.

A namelist specifier cannot be used in a statement that contains a format specifier.

7.1.1.6 Record Specifier

The record specifier identifies the number of the record you wish to access in a file with relative organization. It takes either one of the following forms:

```
REC = r
'r
```

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 records allowed in the file. If necessary, a record number is converted to integer data type before being used.

7.1.1.7 Key-Field-Value Specifier

The key-field-value specifier identifies the key field of a record that you wish to access in an indexed file. The key-field value is equal to the contents of a key field. The key field contains such information as the number, direction, length, byte offset, and type of the fields. It can be used to access records in indexed files because it determines their location.

The attributes of the key field are specified at file creation. Records in an indexed file have the same attributes for their key fields.

Key-field-value specifiers have two components:

- An expression (val) that specifies a value used to compare with key-field values.
- A selection condition keyword (such as KEY) that specifies how to compare val with key-field values.

They take the following forms:

Ascending Keys

```
KEY = val
KEYEQ = val
KEYNXT = val
KEYNXTNE = val
KEYGT = val
KEYGE = val
```

Descending Keys

```
KEY = val
KEYEQ = val
KEYNXT = val
KEYNXTNE = val
KEYLT = val
KEYLE = val
```

val

Is an integer or character expression. Integer expressions must compare with integer key fields; they cannot contain real or complex values. Character expressions must compare with character key fields; they must be ASCII strings made up of either characters or BYTE (LOGICAL*1) array names containing Hollerith data.

The Selection Condition

The selection condition determines how val is compared with key-field values. The keyword can be any one of the following specifiers:

Ascending Keys

- KEY and KEYEQ—the value in the key field must be equal to val.
- KEYNXT—the value in the key field must be the next value of the key equal to or greater than val.
- KEYNXTNE—the value in the key field must be the next value of the key strictly greater than val.
- KEYGT—the value in the key field must be greater than val.
- KEYGE—the value in the key field must be greater than or equal to val.

Descending Keys

- KEY and KEYEQ—the value in the key field must be equal to val.
- KEYNXT—the value in the key field must be the next value of the key equal to or less than val.
- KEYNXTNE—the value in the key field must be the next value of the key that is strictly less than val.
- KEYLT—the value in the key field must be less than val.
- KEYLE—the value in the key field must be less than or equal to val.

Keyword specifiers are interchangeable between ascending-key files and descending-key files—except KEYGT, KEYGE, KEYLT, and KEYLE. These four keywords are one-directional. If you use them with keys going in the opposite direction, the Run-Time Library (RTL) displays an error at run time.

KEYNXT and KEYNXTNE are interchangeable between ascending-key files and descending-key files. Unlike KEY and KEYEQ, they are interpreted differently depending on the direction of the keys in the file. In ascending-key files, KEYNXT is the same as KEYGE, and KEYNXTNE is the same as KEYGT. In descending-key files, KEYNXT is the same as KEYLE, and KEYNXTNE is the same as KEYLT.

You should use KEYGT and KEYGE when exclusively accessing ascending keys and, similarly, KEYLT and KEYLE when exclusively accessing descending keys. When a program must be able to use either kind, you should use KEYNXT and KEYNXTNE.

The Selection Process

To select key-field integer values, the process compares values using the signed integers themselves.

To select key-field character values, the process compares values using the ASCII collating sequence. Additionally, the comparative length of val and a key-field value, and the specified selection condition determine the kind of selection that occurs: exact, generic, or approximate-generic.

Exact selections occur when the expression in val is equal in length to the expression in the key field of the currently accessed record and the keyword specifies a unique selection.

Generic selections occur when the expression in val is shorter than the expression in the key field of the currently accessed record and the keyword specifies a unique selection. The process compares all the characters in val, from left to right, with the same amount of characters in the key field, also from left to right. Remaining key-field characters are ignored.

For example, if val is 'ABCD' and a record's key-field is 10 characters long, and you specified an equal selection, the process could select a record with a key-field value 'ABCDEFGHIJ'.

An approximate-generic selection occurs when val is shorter than the expression in the key field and the keyword (KEYGT, KEYGE, KEYLT, KEYLE, KEYNXT, and KEYNXTNE) does not specify a unique selection. As with generic selections, the process uses only the leftmost characters in

the key-field to compare values. It selects the first key field that satisfies the generic selection criterion.

For example, if val is 'ABCD' and a record's key-field value is 5 characters long and you specify a greater-than selection, the process could select the key-field value 'ABCEx' (and not the key-field value 'ABCDA').

No selection occurs if val is longer than the key-field value. The RTL displays an error message.

7.1.1.8 Key-of-Reference Specifier

The key-of-reference specifier may optionally accompany the key-field-value specifier; it designates the key-field index that is searched to find the specified key-field value. Key-of-reference specifiers take the following form:

```
KEYID=kn
```

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.

7.1.1.9 I/O Status Specifier

The I/O status specifier designates a variable in which is a stored value 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 takes the following form:

```
IOSTAT=ios
```

ios
Is an integer scalar memory reference.

See the *VAX FORTRAN User Manual* for more information on the error numbers returned by IOSTAT.

7.1.1.10 Transfer-of-Control Specifiers

The transfer-of-control specifier identifies an executable statement and transfers control to that statement if an end-of-file or error condition occurs. It takes either one of the following forms:

```
END=s
ERR=s
```

s
Is the label of the executable statement that receives control.

A sequential READ statement can include either or both of the previous 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-of-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 designated by the END=s specification. If there is no END=s specification, control transfers to the statement designated by the ERR=s specification. If there is neither specification (nor an IOSTAT specifier), the program terminates.

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.

See the *VAX FORTRAN User Manual* for a description of 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 7.1.1.9.

Examples

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)
```

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
```

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
```

7.1.2 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 2.2.6 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 takes the following form:

```
s[,s]...
```

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.

7.1.2.1 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 statement 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 referencesare 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 referencesappear in the list.

7.1.2.2 Implied-DO Lists in I/O Statements

An implied-DO list is an I/O list element that acts as though it were a part of an I/O statement within a DO loop. Implied-DO lists can achieve the following:

- 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 takes the following form:

```
(list, i=e1,e2[,e3])
```

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 5.3). 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.

Examples

The following examples demonstrate valid implied-DO lists.

- The following two WRITE statements have the same effect:

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

WRITE (3,200) A,B,C,A,B,C,A,B,C
```

- In the next 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.

- In a series of nested implied-DO lists, parentheses indicate the nesting (see Section 5.3.1.2). Execution of the innermost lists is repeated most often:

```
      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.

- The entire list of an implied-DO list is transmitted before the control variable is incremented:

```
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.

- When processing multidimensional arrays, you can use a combination of fixed subscripts and subscripts that vary according to an implied-DO list:

```
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.

- The value of the control variable can also be output directly:

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

  This statement prints the integers 1 through 20.

If the I/O statement containing an implied-DO list terminates abnormally (with an END= or ERR= transfer or with an IOSTAT value other than zero), the loop control variable becomes undefined.

7.2 READ Statements

The READ statement transfers input data to internal storage from records contained in external logical units or to internal storage from internal files.

VAX FORTRAN provides the following kinds of READ statements:

- Sequential
- Direct
- Internal
- Indexed

7.2.1 Sequential READ Statements

Sequential READ statements transfer input data to internal storage from external records that were sequentially accessed. They can be formatted, list-directed, namelist-directed, or unformatted, taking one of the following forms:

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]
```

Control-list parameters are symbolized as follows:

extu—a logical unit specifier

fmt—a format specifier

f—the nonkeyword form of a format specifier

—a list-directed format specifier (You can also use FMT=.)

nml—a namelist specifier

n—the nonkeyword form of a namelist specifier

iostat—an I/O status specifier

err, end—transfer-of-control specifiers

The I/O-list parameter is symbolized as follows:

iolist—the I/O list specifier

The parameters in I/O statements are fully described in Sections 7.1.1 (control-list parameters) and 7.1.2 (I/O-list parameter). Their syntax rules are summarized in Section 7.1.1.1.

7.2.1.1 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 keyedmode 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 7.2.3 for information about the combined use of formatted sequential READ statements and indexed READ statements under the keyed mode of access

7.2.1.2 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 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 3–1).

  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: .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 must be 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 previously specified.
- 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 record can use any one of the following entities as a value separator, with or without surrounding spaces of tabs:

- Space or tab
- Comma
- Slash

The slash terminates processing of the input statement and the record, leaving all remaining I/O list elements unchanged.

When any of the preceding entities 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 (the last character in the previous record is immediately followed 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 of the values in a record are used, the remainder of the record is ignored.

Example

Consider a program unit with the following statements:

```
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 that will be read:

```
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.

7.2.1.3 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.

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.

Namelist input takes the following form:

```
column 2
↓
$group-name entity = value [,entity = value ,...] $[END]
```

$

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 7.2.1.2).

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

The following syntax rules apply to namelist input:

- The group-name cannot contain spaces or tabs and must be contained within a single record.
- 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.
- 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.
- A logical constant represents true or false values: .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.
- 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.
- 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.
- The form r* indicates r occurrences of a null value, where r is an unsigned integer constant.

- 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.
- The equal sign in a value assignment can be preceded and followed by any number of spaces or tabs.
- 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. If you precede the question mark with an equal sign (=?), the group name and current values of the namelist entities for that group are displayed as in namelist output (see Section 7.3.1.3).

Assigning Values

Input values can be assigned in any order using an assignment of the form: entity=value. Each new line of input can begin in column 2 or in any column thereafter. Column 1 of each record is assumed to contain a FORTRAN carriage-control character. Any data placed in that column is ignored.

Assigned values, array subscripts, and substring specifiers must be constant values. Symbolic (PARAMETER) constants are 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 3–1). Conversion between numeric and character data types is not permitted.

Examples

In the first 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.

```
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 the next example, values are assigned to all of the namelist entities previously associated with the group-name CONTROL.

```
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 as follows:

Entity	Value
TITLE	TESTT002AA
RESET	T
START	10.2
STOP	14.5
INTERVAL	1

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 the following statement:

```
column 2
↓
$CONTROL TITLE(9:10)='BB' $END
```

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. Consider the following example:

A program unit contains the following statements:

```
DIMENSION ARRAY(20)
NAMELIST /ELEM/ ARRAY
READ (UNIT=1,NML=ELEM)
```

The input contains the following:

```
column 2
↓
$ELEM
ARRAY=1.1, 1.2, , 1.4$END
```

Upon program execution, the READ statement assigns values to array elements as follows:

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. In this 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.

7.2.1.4 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 and 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 of 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.

Examples

In the first 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
```

In the second example, the READ statement advances the file connected to logical unit 8 by one record.

```
READ (8)
```

7.2.2 Direct Access READ Statements

Direct access READ statements transfer input data to internal storage from external records accessed under the direct mode of access. They can be formatted or unformatted, taking one of the following forms:

Formatted

```
READ (extu,rec,fmt[,iostat][,err]) [iolist]
```

Unformatted

```
READ (extu,rec[,iostat][,err]) [iolist]
```

Control-list parameters are symbolized as follows:

extu—a logical unit specifier

rec—a record specifier

fmt—a format specifier

iostat—an I/O status specifier

err—a transfer-of-control specifier

The I/O-list parameter is symbolized as follows:

iolist—the I/O-list specifier

The parameters in I/O statements are fully described in Sections 7.1.1 (control-list parameters) and 7.1.2 (I/O-list parameter) and their control-list syntax rules are summarized in Section 7.1.1.1.

7.2.2.1 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 left-to-right order in which the elements appear.

If the I/O list and formatting do not use all of 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.

Example

In the following 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.

```
      READ (2,REC=35,FMT=10) (NUM(K), K=1,10)
10    FORMAT (10I2)
```

7.2.2.2 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

In the first 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)
```

In the second 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)
```

7.2.3 Indexed READ Statements

Indexed READ statements transfer input data to internal storage from external records using keyed access.

In an indexed file, a series of records can be read in key value sequence by using an indexed READ statement together with a sequential READ statement. The first record in the sequence is read using the indexed statement and the rest are read using sequential READ statements.

Indexed READ statements can be formatted or unformatted, taking one of the following forms:

Formatted

```
READ (extu,fmt,key[,keyid][,iostat][,err]) [iolist]
```

Unformatted

```
READ (extu,key[,keyid][,iostat][,err]) [iolist]
```

Control-list parameters are symbolized 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 symbolized as follows:

iolist—the I/O-list specifier

All of the parameters used in I/O statements are described in Sections 7.1.1 (control-list parameters) and 7.1.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 7.1.1.1.

7.2.3.1 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 can 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.

Example

In the following 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.

```
READ (3,KAT(25),KEY='ABCD') A,B,C,D
```

7.2.3.2 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 that is referred to, an error occurs.

Examples

In the first 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.

```
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 the second 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.

```
READ (3,KEYGE='XYZDEF',KEYID=2,ERR=99) IKEY
```

7.2.4 Internal READ Statement

Internal READ statements transfer input data to internal storage from an internal file. (This statement has an alternative statement, DECODE, which is discussed in Appendix A.)

The internal READ statement can be formatted or list-directed, taking one of the following forms:

Formatted

```
READ (intu,fmt[,iostat][,err][,end]) [iolist]
```

List-Directed

```
READ (intu,*[,iostat][,err][,end]) [iolist]
```

Control-list parameters are symbolized as follows:

intu—an internal file specifier

fmt—a format specifier

—a 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 symbolized as follows:

iolist—the I/O-list specifier

The parameters used in I/O statements are described in Sections 7.1.1 (control-list parameters) and 7.1.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 7.1.1.1.

7.2.4.1 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 left-to-right order in which the elements appear.

Example

The following 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.

```
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
```

7.2.4.2 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 the *VAX FORTRAN User Manual* for information on the characteristics and use of internal files.

7.3 WRITE Statements

The WRITE statement transfers output data from internal storage to user-specified external logical units (disks, printers, terminals, mailboxes) or to internal files. It can be used in sequential, direct, keyed, or internal access modes.

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 7.4.

7.3.1 Sequential WRITE Statements

Sequential WRITE statements transfer output data from internal storage to external records accessed under the sequential mode of access. (See the *VAX FORTRAN User Manual* for descriptions of the various access modes.)

Sequential WRITE statements can be formatted, list-directed, namelist-directed, or unformatted, taking one of the following forms:

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]
```

Control-list parameters are symbolized as follows:

extu—a logical unit specifier

fmt—a format specifier

—a 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 symbolized as follows:

iolist—the I/O-list specifier

All of the parameters used in I/O statements are described in Sections 7.1.1 (control-list parameters) and 7.1.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 7.1.1.1.

7.3.1.1 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 that is sequentially accessed.

The length of the records written to a user-specified output device (for example, a line printer) must not exceed the maximum record length that the 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.

If you transfer numeric data using formatted output statements and you subsequently use the data as input, the resulting data may not be precise because of a rounding of the data during conversion from binary to character form. Therefore, if you expect to subsequently use numeric data as input, use unformatted output and input statements for data transfer.

Examples

In the first 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')
```

In the second 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)
```

In the third example, the WRITE statement writes three separate records to logical unit 1. Each record has only one field.

```
      WRITE (1,900) DEE, EEE, EFF
900   FORMAT (F8.5)
```

7.3.1.2 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 constant values transferred as input by the list-directed READ and ACCEPT statements, with the following exceptions:

- Character constants are transferred without delimiting apostrophes.
- Each internal apostrophe is represented by only one apostrophe instead of two.

Consequently, records containing list-directed character output data can be printed but not used for list-directed input. (See Section 7.2.1.2 for a discussion of list-directed value forms.)

Table 7–1 shows the default output formats for each data type.

Table 7–1: List-Directed Default 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

List-directed output formats behave in the following ways:

- 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.

Example

The following example illustrates a valid list-directed WRITE statement:

```
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
```

7.3.1.3 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.

Example

Consider a program unit with the following statements:

```
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)
    .
    .
    .
```

The input contains the following statements:

```
Δ$PARAMΔNAME(2)(10:)='HEISENBERG',
ΔPITCH=5.0, YAW=0.0, ROLL=5.0,
ΔDIAGNOSTICS=.TRUE.
ΔITERATIONS=10$END
```

In this case, the WRITE statement would write the following:

```
Δ$PARAM
ΔNAME   = '                          ', '        HEISENBERG',
ΔPITCH  =    5.000000    ,
ΔROLL   =    5.000000    ,
ΔYAW    =   0.0000000E+00,
ΔPOSITION       = 3*0.0000000E+00,
ΔDIAGNOSTICS    = T,
ΔITERATIONS     =                10
Δ$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.

7.3.1.4 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.

Examples

In the first 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 (1) (LIST(K), K=1,5)
```

In the second example, the WRITE statement writes one null record to the file connected to logical unit 4.

```
WRITE (4)
```

7.3.2 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 attributes of a direct access file are established by the OPEN statement.)

Direct access WRITE statements can be formatted or unformatted, taking one of the following forms:

Formatted

```
WRITE (extu,rec,fmt[,iostat][,err]) [iolist]
```

Unformatted

```
WRITE (extu,rec[,iostat][,err]) [iolist]
```

Control-list parameters are symbolized 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 symbolized as follows:

iolist—the I/O-list specifier

The parameters used in I/O statements are described in Sections 7.1.1 (control-list parameters) and 7.1.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 7.1.1.1.

7.3.2.1 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 overfill the record, an error occurs.

7.3.2.2 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.

7.3.3 Indexed WRITE Statements

The indexed WRITE statement transfers output data from internal storage to external records accessed under the keyed mode of access. (The OPEN statement establishes the attributes of an indexed file.)

Indexed WRITE statements always write a new record. You should use the REWRITE statement to update an existing record (see Section 7.4).

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.

Indexed WRITE statements can be formatted or unformatted, taking one of the following forms:

Formatted

```
WRITE (extu,fmt[,iostat][,err]) [iolist]
```

Unformatted

```
WRITE (extu[,iostat][,err]) [iolist]
```

Control-list parameters are symbolized as follows:

extu—a logical unit specifier

fmt—a format specifier

iostat—an I/O status specifier

err—a transfer-of-control specifier

The I/O-list parameter is symbolized as follows:

iolist—the I/O-list specifier

The parameters used in I/O statements are described in Sections 7.1.1 (control-list parameters) and 7.1.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 7.1.1.1.

7.3.3.1 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.

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.

Example

In the following example, the formatted indexed 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. KEYVAL is the key by which the record is accessed. (This example assumes that the first 10 bytes of a record are a character key.)

```
      WRITE (4,100) KEYVAL, (RDATA(I), I=1,20)
100   FORMAT (A10,20F15.7)
```

7.3.3.2 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 overfill the record, an error occurs.

Using records (structured data items) has advantages when writing to indexed files. Such files usually have a fixed record format. Thus, by using a structure declaration that models the file's record format, you can accomplish the I/O operation with a single record variable instead of a potentially long I/O list. For an example, refer to the *VAX FORTRAN User Manual*.

7.3.4 Internal WRITE Statement

Internal WRITE statements transfer output data from internal storage to an internal file. (You can also use the ENCODE statement discussed in Appendix A to control internal output.) See the *VAX FORTRAN User Manual* for information about the characteristics and use of internal files.

Namelist-directed formatting is not permitted with internal WRITE statements.

Internal write statements can be formatted or list-directed, taking one of the following forms:

Formatted

```
WRITE (intu,fmt[,iostat][,err]) [iolist]
```

List-Directed

```
WRITE (intu,*[,iostat][,err]) [iolist]
```

Control-list parameters are symbolized as follows:

intu—an internal file specifier

fmt—a format specifier

—a list-directed formatting specifier (You can also use FMT=.)

iostat—an I/O status specifier

err—a transfer-of-control specifier

The I/O-list parameter is symbolized as follows:

iolist—the I/O-list specifier

The parameters used in I/O statements are described in Sections 7.1.1 (control-list parameters) and 7.1.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 7.1.1.1.

7.3.4.1 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.

7.3.4.2 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.

7.4 REWRITE Statement

The REWRITE statement transfers data from internal storage to the current record in a file with indexed or relative organization. The *current record* is the one that was most recently accessed by a successful direct access, indexed, or sequential READ statement.

Between a READ and REWRITE statement, you should not issue any other I/O statement (except INQUIRE) on that logical unit. Execution of any other I/O statement on the logical unit destroys the current-record context and causes the current record to become undefined.

REWRITE statements can be formatted or unformatted, taking one of the following forms:

Formatted

```
REWRITE (extu,fmt[,iostat][,err]) [iolist]
```

Unformatted

```
REWRITE (extu[,iostat][,err]) [iolist]
```

Control-list parameters are symbolized as follows:

extu—a logical unit specifier

fmt—a format specifier

iostat—an I/O status specifier

err—a transfer-of-control specifier

The I/O list parameter is symbolized as follows:

iolist—an I/O-list specifier

The parameters used in I/O statements are described in Sections 7.1.1 (control-list parameters) and 7.1.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 7.1.1.1.

7.4.1 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 formatted REWRITE statement writes to the current record in the file (the last record accessed by a preceding direct access, indexed, or sequential READ statement).

Errors occur with the following conditions:

- If the primary key value is changed, an error usually occurs.
- If you attempt to rewrite more than one record in a single REWRITE statement operation, an error occurs.
- If a record is too long, an error occurs. (A record might become too long when unused space in a rewritten fixed-length record is filled with spaces.)

Example

In the following 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.

```
     REWRITE (3,10,ERR=99) NAME, AGE, BIRTH
10   FORMAT (A16,I2,A8)
```

7.4.2 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 unformatted REWRITE statement writes to the current record in the file (the last record accessed by a preceding indexed or sequential READ statement). Unused space in a rewritten fixed-length record is filled with zeros.

Errors occur with the following conditions:

- If the primary key value is changed, an error usually occurs.
- If you attempt to rewrite more than one record in a single REWRITE statement operation, an error occurs.
- If a record is too long, an error occurs.

7.5 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.

ACCEPT statements take any one of the following forms:

```
ACCEPT f[,iolist]

ACCEPT *[,iolist]

ACCEPT n
```

Control-list parameters are symbolized as follows:

f—the nonkeyword form of a format specifier

*—a list-directed formatting specifier

n—the nonkeyword form of a namelist specifier

The I/O-list parameter is symbolized as follows:

iolist—an I/O list specifier

The parameters used in I/O statements are described in Sections 7.1.1 (control-list parameters) and 7.1.2 (I/O-list parameter). The rules for specifying control-list parameters are summarized in Section 7.1.1.1.

The ACCEPT statement functions exactly like the sequential READ statements discussed in Sections 7.2.1.1 through 7.2.1.3, with the following important exception: the ACCEPT statement can never be connected to user-specified logical units.

Example

In the following 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:

```
      CHARACTER*10 CHARAR(5)
      ACCEPT 200, CHARAR
200   FORMAT (5A10)
```

7.6 TYPE and PRINT Statements

The TYPE and PRINT statements transfer output data from internal storage to external records that are sequentially accessed.

TYPE and PRINT statements take the same forms, which can be any one of the following:

```
TYPE f[,iolist]
PRINT f[,iolist]

TYPE *[,iolist]
PRINT *[,iolist]

TYPE n
PRINT n
```

Control-list parameters are symbolized 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 symbolized as follows:

iolist—the I/O-list specifier

The parameters used in I/O statements are described in Sections 7.1.1 (control-list parameters) and 7.1.2 (I/O-list parameter).

TYPE and PRINT statements function exactly like the formatted sequential WRITE statement discussed in Section 7.3.1.1, with the following important exception: the formatted sequential TYPE and PRINT statements can never be used to transfer data to user-specified logical units.

Example

In the following example, the PRINT statement writes one record to the implicit output device. The record has four fields of character data.

```
      CHARACTER*16 NAME, JOB
      PRINT 400, NAME, JOB
400   FORMAT ('NAME=',A,'JOB=',A)
```

Chapter 8

I/O Formatting

Formatting I/O statements specify the form of data being transferred. They also specify the data conversion (editing) required to achieve that form. The primary formatting statement is the FORMAT statement. It is a nonexecutable statement used in conjunction with formatted I/O statements and with ASSIGN, ENCODE, and DECODE statements.

This chapter contains the following information about I/O formatting:

- General rules for writing FORMAT statements (Section 8.1)
- FORMAT statement syntax (Section 8.2)
- Field and edit descriptors (Section 8.3)
- Carriage control options for output records (Section 8.4)
- Format specifications and field separators (comma and slash) (Sections 8.5 and 8.6)
- Run-time formats (instead of a FORMAT statement) that dynamically creates formats during program execution (Section 8.7)
- Format control interactions with I/O lists (Section 8.8)

8.1 General Rules for Writing FORMAT Statements

This section summarizes the rules for constructing and using the format specifications and their components in FORMAT statements. It also summarizes the rules for constructing external fields and records.

The following are general FORMAT statement rules:

- A FORMAT statement must always be labeled.
- 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.
- 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.
- In a field descriptor such as nHc1c2 ... cn, exactly n characters must follow the H format code. You can use any printable ASCII character in this field descriptor.
- In a scale factor of the form nP, n must be an integer constant or variable format expression in the range –128 to 127. 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.
- 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.
- 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.

- 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.
- If a character-constant format includes apostrophes, those apostrophes must be represented by double apostrophes (").

8.1.1 Input Rules for FORMAT Statements

The following are general FORMAT statement input rules:

- A minus sign must precede a negative value in an external input field; a plus sign is optional before a positive value.
- 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.
- 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; an E(e), D(d), or Q(q) exponent field; or both.
- 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.
- 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.
- 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, exponent, or combination of the three).
- 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.

8.1.2 Output Rules for FORMAT Statements

The following are general FORMAT statement output rules:

- 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.
- 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.
- 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.

8.2 FORMAT Statement Syntax

The FORMAT statement takes the following form:

```
FORMAT (q1f1s1f2s2 ... fnqn)
```

qn
Is zero or more slash (/) record terminators. (See Section 8.5.)

fn
Is a field descriptor or a group of field descriptors enclosed in parentheses. (See Section 8.3.)

s
Is a field separator. (See Sections 8.5 and 8.6.)

The entire list of field descriptors and field separators, including the parentheses, is called the format specification.

The field descriptor takes one of the following forms:

```
[r]c
[r]cw
[r]cw.m
[r]cw.d[Ee]
```

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 r, w, m, and d terms 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 with some field descriptors but invalid with others (see Section 8.3.1).

The d and e terms are required with some field descriptor but invalid with others.

PARAMETER constants cannot be used as values for the r, w, m, d, or e specifiers.

The field descriptors are as follows:

- 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).

Table 8–1 summarizes the FORMAT codes.

Table 8–1: FORMAT Code Summary

Code	Form	Effect
A	A[w]	Transfers character or Hollerith values. (See Section 8.3.9.1.)
BN	BN	Specifies that embedded and trailing blanks in a numeric input field are to be ignored. (See Section 8.3.3.1.)
BZ	BZ	Specifies that embedded and trailing blanks in a numeric input field are to be treated as zeros. (See Section 8.3.3.2.)
D	Dw.d	Transfers real values (D exponent field indicator). (See Sections 8.3.6.3 and 8.3.6.5.)
E	Ew.d[Ee]	Transfers real values (E exponent field indicator). (See Sections 8.3.6.2 and 8.3.6.5.)
F	Fw.d	Transfers real values. (See Sections 8.3.6.2 and 8.3.6.5.)
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. (See Sections 8.3.6.4 and 8.3.6.5.)
H	nHc...c	Transfers data between the H field descriptor and an external record. (See Section 8.3.9.2.)
I	Iw[.m]	Transfers decimal integer values. (See Section 8.3.5.1.)
L	Lw	Transfers logical data: on input, transfers characters; on output, transfers T or F. (See Section 8.3.8.)
O	Ow[.m]	Transfers octal values. (See Section 8.3.5.2.)

Table 8–1 (Cont.): FORMAT Code Summary

Code	Form	Effect
P	nP	Alters locations of decimal points. (See Section 8.3.7.)
Q	Q	Obtains the number of characters remaining to be transferred in an input record. (See Section 8.3.12.1.)
S	S	Reinvokes optional plus characters in numeric output fields; counters the action of SP and SS. (See Section 8.3.4.3.)
SP	SP	Writes plus characters that would otherwise be optional into numeric output fields. (See Section 8.3.4.1.)
SS	SS	Suppresses optional plus characters in numeric output fields. (See Section 8.3.4.2.)
T	Tn	Specifies positional tabulation. (See Section 8.3.11.2.)
TL	TLn	Specifies relative tabulation (left). (See Section 8.3.11.3.)
TR	TRn	Specifies relative tabulation (right). (See Section 8.3.11.4.)
X	nX	Specifies that n characters are to be skipped. (See Section 8.3.11.1.)
Z	Zw[.m]	Transfers hexadecimal values. (See Section 8.3.5.3.)
$	$	Suppresses carriage return during interactive I/O. (See Section 8.3.12.2.)
:	:	Terminates format control if the I/O list is exhausted. (See Section 8.3.12.3.)

8.3 Field and Edit Descriptors

A field descriptor describes the size and format of one or more 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. Default values are as follows:

- BLANK = 'NULL'—when an open has been executed
- BLANK = 'ZERO'—when no explicit open has been executed

During 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 8.3.3 through 8.3.12 describe each of the field and edit descriptors.

8.3.1 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 are prohibited). 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 8.8 discusses how to use parentheses when the number of values to be formatted is greater than the number of format specifications.

8.3.2 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.

Syntax Rules

The following syntax rules apply to 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 8.8 for a description of the synchronization of I/O lists with formats.

Example

Consider the following statements:

```
      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
```

8.3.3 Blank Control Editing

The treatment of embedded and trailing blanks within numeric input files is controlled by BN and BZ edit descriptors.

8.3.3.1 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 takes the following 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.

8.3.3.2 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 takes the following form:

```
BZ
```

The BZ descriptor affects only I, O, Z, F, E, D, and G editing during the execution of an input statement.

8.3.4 Sign Control Editing

The treatment of optional plus characters in output data is controlled by SP, SS, and S edit descriptors.

8.3.4.1 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 takes the following form:

```
SP
```

The SP descriptor affects only I, F, E, D, and G editing during the execution of an output statement.

8.3.4.2 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 previously. The SS descriptor takes the following form:

```
SS
```

The SS descriptor affects only I, F, E, D, and G editing during the execution of an output statement.

8.3.4.3 S Edit Descriptor

The S edit descriptor reinvokes optional plus characters (+) in numeric output fields. It takes the following 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.

8.3.5 Integer Editing

Integer editing is controlled by I (decimal), O (octal), and Z (hexadecimal) field descriptors.

8.3.5.1 I Field Descriptor

The I field descriptor transfers decimal integer values. It takes the following form:

```
Iw[.m]
```

The corresponding I/O list element must have an 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.

The I field descriptor processes input in the following ways:

- 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.
- If the field is blank, it is treated as a value of zero.

The following list illustrates valid input processing with the I field descriptor:

Format	External Field	Internal Value
I4	2788	2788
I3	-26	-26
I9	ΔΔΔΔΔΔ312	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.

The I field descriptor processes output in the following ways:

- 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. Therefore, the term w must be large enough to fit a minus sign when necessary.
- If m is present, the external field consists of at least m digits and is filled with zeros on the left, if necessary.

The following list illustrates valid output processing with the I field descriptor:

Format	Internal Value	External Representation
I3	284	284
I4	-284	-284
I4	0	ΔΔΔ0
I5	174	ΔΔ174
I2	3244	**
I3	-473	***
I7	29.812	(Not permitted: error)
I4.0	0	ΔΔΔΔ
I4.2	1	ΔΔ01
I4.4	1	0001

If m is zero and the internal representation is zero, the external field is blank.

8.3.5.2 O Field Descriptor

The O field descriptor transfers octal (base 8) values and can be used with any data type. It takes the following 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. An error occurs if the value of the external field exceeds the range of the corresponding list element.

The following list illustrates valid input processing using the O field descriptor:

Format	External Field	Internal Octal Value
O5	32767	32767
O4	16234	1623
O3	97Δ	(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.

The following list illustrates valid output processing using O field descriptors:

Format	Internal (Decimal) Value	External Representation
O6	32767	Δ77777
O6	-32767	100001
O2	14261	**
O4	27	ΔΔ33
O5	10.5	41050
O4.2	7	ΔΔ07
O4.4	7	0007

If m is zero and the external representation is zero, the external field is filled with blanks.

8.3.5.3 Z Field Descriptor

The Z field descriptor transfers hexadecimal (base 16) values, and can be used with any data type. It takes the following 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, decimal point, or 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.

The following list illustrates valid input processing using the Z field descriptor:

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 filled with zeros on the left, if necessary.

The following list illustrates valid output processing using the the Z field descriptor:

Format	Internal (Decimal) Value	External Representation
Z4	32767	7FFF
Z5	-32767	Δ8001
Z2	16	10
Z4	-10.5	C228
Z3.3	2708	A94
Z6.4	2708	ΔΔ0A94

If m is zero and the internal representation is zero, the external field is filled with blanks.

8.3.6 Real Editing

Editing performed on data with a real data type is controlled by the F, E, D, and G field descriptors.

NOTE

When attempting to parse textual input, you should not mix 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, +, –, and . are all treated as 0.0.

8.3.6.1 F Field Descriptor

The F field descriptor transfers real values. It takes the following form:

```
Fw.d
```

The corresponding I/O list element must have a 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.

Input processing with the F field descriptor behaves in the following ways:

- 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.
- If a field is all blank, it is treated as a zero value.
- If a field contains only an exponent or decimal point, it is treated as a zero value.
- 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.

The following list illustrates valid input processing using the F field descriptor:

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

- Minus sign when necessary (plus signs are optional)

- At least one digit to the left of the decimal point
- Decimal point
- d digits to the right of the decimal

In other words, w must be greater than or equal to d+3.

The following list illustrates valid output processing using the F field descriptor:

Format	Internal Value	External Representation
F8.5	2.3547188	Δ2.35472
F9.3	8789.7361	Δ8789.736
F2.1	51.44	**
F10.4	-23.24352	ΔΔ-23.2435
F5.2	325.013	******
F5.2	-.2	-0.20

8.3.6.2 E Field Descriptor

The E field descriptor transfers real values in exponential form. It takes the following form:

```
Ew.d[Ee]
```

The corresponding I/O list element must have a 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.

The following example illustrates valid input processing using the E field descriptor:

Format	External Field	Internal Value
E9.3	734.432E3	734432.0
E12.4	ΔΔ1022.43E-6	1022.43E-6
E15.3	52.3759663ΔΔΔΔΔ	52.3759663
E12.5	210.5271D+10	210.5271E10

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 into *standard form*. The standard form has the following components:

- Minus sign, when necessary (plus signs are optional)
- Zero
- Decimal point
- d digits to the right of the decimal point
- e + 2-character exponent that takes one of the following forms:

  For exponents less than or equal to 99, with `Ew.d`:

  ```
  E+nn
  E-nn
  ```

  For exponents less than or equal to 999, with `Ew.d`:

  ```
  +nnn
  -nnn
  ```

  For all exponents with `Ew.dEe`:

  ```
  E+n1n2...ne
  E-n1n2...ne
  ```

The exponent field width specification is optional. If you omit it, 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:

- Minus sign when necessary (plus signs are optional)
- Zero
- Decimal point
- d digits
- Exponent

In other words, w must be greater than or equal to d+7. If e is present, w must be greater than or equal to d+e+5.

The following list illustrates valid output processing using the E field descriptor:

Format	Internal Value	External Representation
E9.2	475867.222	Δ0.48E+06
E12.5	475867.222	Δ0.47587E+06
E12.3	0.00069	ΔΔΔ0.690E-03
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****
E14.5E4	-1.001	-0.10010E+0001
E14.3E6	0.000123	Δ0.123E-000003

8.3.6.3 D Field Descriptor

The D field descriptor transfers real values in exponential form. It takes the following form:

```
Dw.d
```

The corresponding I/O list element must have a 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.

The following list illustrates valid input processing using the D field descriptor:

Format	External Field	Internal Value
BZ,D10.2	12345ΔΔΔΔΔ	12345000.0D0
D10.2	ΔΔ123.45ΔΔ	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.

The following list illustrates valid output processing using the D field descriptor:

Format	Internal Value	External Representation
D14.3	0.0363	ΔΔΔΔΔ0.363D-01
D23.12	5413.87625793	ΔΔΔΔΔ0.541387625793D+04
D9.6	1.2	********

8.3.6.4 G Field Descriptor

The G field descriptor transfers real values in a form that, in effect, combines the F and E field descriptors. It takes the following 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 8–2.

Table 8–2: 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:

- Minus sign when necessary (plus signs are optional)
- Decimal point
- One digit to the left of the decimal point
- d digits to the right of the decimal point
- Either a 4-character or e+2–character exponent

In other words, w must be greater than or equal to d+8. If e is present, w must be greater than or equal to d+e+6.

The following list illustrates valid output processing using the G field descriptor:

Format	Internal Value	External Representation
G13.6	0.01234567	Δ0.123457E-01
G13.6	-0.12345678	-0.123457ΔΔΔΔ
G13.6	1.23456789	ΔΔ1.23457ΔΔΔΔ
G13.6	12.34567890	ΔΔ12.3457ΔΔΔΔ
G13.6	123.45678901	ΔΔ123.457ΔΔΔΔ
G13.6	-1234.56789012	Δ-1234.57ΔΔΔΔ
G13.6	12345.67890123	ΔΔ12345.7ΔΔΔΔ
G13.6	123456.78901234	ΔΔ123457.ΔΔΔΔ
G13.6	-1234567.89012345	-0.123457E+07

The following list shows the same values output-processed as the previous list, except the following list uses an equivalent F field descriptor:

Format	Internal Value	External Representation
F13.6	0.01234567	ΔΔΔΔΔ0.012346
F13.6	-0.12345678	ΔΔΔΔ-0.123457
F13.6	1.23456789	ΔΔΔΔΔ1.234568
F13.6	12.34567890	ΔΔΔΔ12.345679
F13.6	123.45678901	ΔΔΔ123.456789
F13.6	-1234.56789012	Δ-1234.567890
F13.6	12345.67890123	Δ12345.678901
F13.6	123456.78901234	123456.789012
F13.6	-1234567.89012345	**********

8.3.6.5 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.

The following list illustrates valid input processing using complex data editing:

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.

The following list illustrates valid output processing using complex data editing.

Format	Internal Value	External Representation
2F8.5	2.3547188, 3.456732	Δ2.35472 Δ3.45673
E9.2,'Δ,Δ',E5.3	47587.222, 56.123	Δ0.48E+06Δ,Δ*****

8.3.7 Scale Factor Editing—P Edit Descriptor

During either input or output processing, the scale factor lets you alter the location of the decimal point in real values and in the two parts of complex values. The scale factor takes the following form:

nP

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 it must precede the first field descriptor that associates with it; for example:

```
nPFw.d     nPEw.d     nPDw.d     nPGw.d
```

Input Processing

On input, the scale factor associated with an F, E, D, or G field descriptor 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.

The following list illustrates valid input processing using scale factor editing:

Format	External Field	Internal Value
3PE10.5	ΔΔΔ37.614Δ	.037614
3PE10.5	ΔΔ37.614E2	3761.4
-3PE10.5	ΔΔΔΔ37.614	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.

The following list illustrates valid output processing using scale factor editing:

Format	Internal Value	External Representation
1PE12.3	-270.139	ΔΔ-2.701E+02
1PE12.2	-270.139	ΔΔΔ-2.70E+02
-1PE12.2	-270.139	ΔΔΔ-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. However, once you specify a scale factor, it applies to all subsequent real field descriptors in the same FORMAT statement, unless another scale factor appears. For example, consider the following statements:

```
      DIMENSION A(6)
      DO 10 I=1,6
10    A(I) = 25.
      TYPE 100,A
100   FORMAT(' ',F8.2,2PF8.2,F8.2)
```

These statements produce the following results:

```
   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 8.8). A scale factor of zero can be reinstated only by an explicit 0P specification.

8.3.8 Logical Editing—L Edit Descriptor

The L field descriptor transfers logical data. It takes the following form:

```
Lw
```

The corresponding I/O list element must have an integer or logical data type.

Input Processing

In an input statement, the L field descriptor transfers w characters from the external field in the following ways:

- 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.
- If the field contains all blanks, the value .FALSE. is assigned.
- If any other value is in the external field, an error occurs.

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.

The following list illustrates valid output processing using logical editing:

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

8.3.9 Character Editing

Editing data with a character data type is controlled by the A and H field descriptors.

8.3.9.1 A Field Descriptor

The A field descriptor transfers character or Hollerith values. It takes the following form:

```
A[w]
```

The corresponding I/O list element can have any data type. If it has a character data type, character data is transmitted. If it has 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 listed in Table 8–3.

Table 8–3: Size Limit of Numeric Elements Using the A Field Descriptor

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 8.3.6.5.

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.

The following list illustrates valid input processing using the A field descriptor:

Format	External Field	Internal Representation	Data Type
A6	PAGEΔ#	#	CHARACTER*1
A6	PAGEΔ#	EΔ#	CHARACTER*3
A6	PAGEΔ#	PAGEΔ#	CHARACTER*6
A6	PAGEΔ#	PAGEΔ#ΔΔ	CHARACTER*8
A6	PAGEΔ#	#	LOGICAL*1
A6	PAGEΔ#	Δ#	INTEGER*2
A6	PAGEΔ#	GEΔ#	REAL
A6	PAGEΔ#	PAGEΔ#ΔΔ	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.

The following list illustrates valid output processing using the A field descriptor:

Format	Internal Value	External Representation
A5	OHMS	ΔOHMS
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 has a character data type, the default value is the length of the I/O list element. If the I/O list element has a numeric data type, the default value is the maximum number of characters that can be stored in a variable of that data type.

8.3.9.2 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:

```
nHc1c2c3 ... cn
```

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.

8.3.9.3 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''SΔDATEΔIS:Δ',I2,'/',I2,'/',I2)
```

When you use a pair of apostrophes this way, they count as a single character.

8.3.10 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 8–4 lists the default values for w, d, and e.

Table 8–4: 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*4,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		

For the A field descriptor, the default is the actual length of the corresponding I/O list element.

8.3.11 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.

8.3.11.1 X Edit Descriptor

The X edit descriptor is a positional specifier. It takes the following form:

```
nX
```

The term n specifies how many character positions are 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 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 ('1PAGEΔNUMBERΔ',I2,16X,'GRAPHICΔANALYSIS,ΔCONT.')
```

The preceding WRITE statement would print a record similar to the following output:

```
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 8.4 describes printer carriage control.)

A trailing X format on a record does not write any characters unless it is followed by another field that does; for example:

```
    WRITE (6,99) K
99  FORMAT ('ΔK=',I6,5X)
```

The preceding example writes a record of only 9 characters. To cause n trailing blanks to be written at the end of a record, use the format n('Δ').

8.3.11.2 T Edit Descriptor

The T edit descriptor is a positional tabulation specifier. It takes 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 nth character position. For example, if an input statement reads a record containing ABCΔΔΔXYZ, and this record is controlled by the following 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 begins at the nth 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 8.4). For example:

```
    PRINT 25
25  FORMAT (T51,'COLUMN 2',T21,'COLUMN 1')
```

These statements print the following line (assuming normal carriage control processing):

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

8.3.11.3 TL Edit Descriptor

The TL edit descriptor is a relative tabulation specifier. It takes the following form:

```
TLn
```

n
Indicates that the next character to be transferred to or from a record is the nth 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.

8.3.11.4 TR Edit Descriptor

The TR edit descriptor is also a relative tabulation specifier. It takes the following form:

```
TRn
```

n
Indicates that the next character to be transferred to or from a record is the nth character to the right of the current character. The value of n must be greater than or equal to one.

8.3.12 Additional Editing Operations

Additional edit descriptors 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.

8.3.12.1 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 takes 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.

8.3.12.2 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 is in effect (see Section 8.4).

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.

Consider the following statements:

```
     TYPE 100
100  FORMAT (' ENTER RADIUS VALUE ',$)
     ACCEPT 200, RADIUS
200  FORMAT (F6.2)
```

The resulting formatted message would appear as follows:

```
ENTER RADIUS VALUE
```

Your response (for example, "12.") can then go on the same line:

```
ENTER RADIUS VALUE   12.
```

8.3.12.3 Colon Descriptor

In a format specification, the colon character (:) 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 8.8 describes format control in detail.

8.4 Carriage Control

Whenever the default for the OPEN statement's CARRIAGECONTROL keyword is in effect ('FORTRAN'), the first character of every record is not printed when it is transferred to a printer. 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 8–5 lists these characters and their effects.

Table 8–5: 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
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 8–5 is treated as a space and is deleted from the print line. If you accidentally omit the carriage control character, the first character of the record is not printed.

8.5 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)
```

Multiple slashes cause input records to be bypassed or blank records to be outputted. 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 previous statements produce the following output:

```
Column 50,      top of page
                ↓
                HEADING LINE
(blank line)
                SUBHEADING LINE
(blank line)
(blank line)
```

8.6 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 reads 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 occur:

I = 1

J = –2

A = 1.0

B = 0.35

The physical end of the record also serves as a field terminator. The d part of a w.d specification is not affected by an external field separator.

A comma can only terminate 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.

A comma cannot 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.

8.7 Run-Time Format

You can store format specifications in character scalar references, numeric array references, or numeric scalar field references (see Section 2.2.5.1). 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. Opening and closing parentheses are required. Variable format expressions are not permitted in run-time formats.

Example

In the following example, the DATA statement assigns a left parenthesis to the character array element FORCHR(0) and 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 be subsequently unavailable for using that array as a run-time format specification.

8.8 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 are 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.

Example

The following annotated example shows several interactions between the I/O list and the FORMAT statement.

Consider a data file named FOR002.DAT:

```
$ TYPE FOR002.DAT
001 0101 0102 0103 0104 0105
002 0201 0202 0203 0204 0205
003 0301 0302 0303 0304 0305
004 0401 0402 0403 0404 0405
005 0501 0502 0503 0504 0505
006 0601 0602 0603 0604 0605
007 0701 0702 0703 0704 0705
008 0801 0802 0803 0804 0805
009 0901 0902 0903 0904 0905
010 1001 1002 1003 1004 1005
$
```

Assume that the data is to be processed 2 records at a time. Each record starts with a number to be put into an element of a vector B, followed by 5 numbers to be put in a row in matrix A.

The following program uses several different FORMAT statements to read the data in FOR002.DAT:

```
      INTEGER I, J, A(2,5), B(2)

      READ (2,100) (B(I), (A(I,J), J=1,5),I=1,2) ❶
100   FORMAT (2 (I3, X, 5(I4,X), /) ) ❷

      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2) ❸
999   FORMAT (' B is ', 2(I3, X), ';  A is', /
     1       (' ', 5 (I4, X)) )

      READ (2,200) (B(I), (A(I,J), J=1,5),I=1,2) ❹
200   FORMAT (2 (I3, X, 5(I4,X), :/) )

      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2) ❺

      READ (2,300) (B(I), (A(I,J), J=1,5),I=1,2) ❻
300   FORMAT ( (I3, X, 5(I4,X)) )

      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2) ❼

      READ (2,400) (B(I), (A(I,J), J=1,5),I=1,2) ❽
400   FORMAT ( I3, X, 5(I4,X) )

      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2) ❾

      END
```

Notes:

❶ This starts by reading B(1); then A(1,1) through A(1,5); then B(2) and A(2,1) through A(2,5).

The first record (starting with 001) is read to start the processing of the I/O list.

❷ There are two records, each in the format I3, X, 5(I4, X). The / forces the reading of the second record after A(1,5) is processed; it also forces the reading of the third record after A(2,5) is processed—no data is taken from that record.

❸ This will output:

```
B is   1   2 ;  A is
 101  102  103  104  105
 201  202  203  204  205
```

❹ This starts by reading the record starting with 004. The / forces the reading of the next record after A(1,5) is processed; the : stops the reading after A(2,5) is processed but before the / forces another read.

❺ This will output:

```
B is   4   5 ;  A is
 401  402  403  404  405
 501  502  503  504  505
```

❻ This starts by reading the record starting with 006. After A(1,5) is processed, format reversion reads the next record and starts format processing at the (before the I3.

❼ This will output:

```
B is   6   7 ;  A is
 601  602  603  604  605
 701  702  703  704  705
```

❽ This starts by reading the record starting with 008. After A(1,5) is processed, format reversion reads the next record and starts format processing at the (before the I4.

❾ This will output:

```
B is   8  90 ;  A is
 801  802  803  804  805
 9010 9020 9030 9040  100
```

The record 009 0901 0902 0903 0904 0905 was processed with I4 as "009 " for B(2), which is 90; X skips the next "0"; then "901 " is processed for A(2,1), which is 9010; "902 " for A(2,2); "903 " for A(2,3); and "904 " for A(2,4). The repetition factor of 5 is now exhausted and the format ends. Format reversion reads another record and starts format processing at the (before the I4 so that "010 " is read for A(2,5), which is 100.

Chapter 9

Auxiliary I/O Statements

The following auxiliary I/O statements manage files during I/O operations:

- OPEN—connects a FORTRAN logical unit to a file or device; declares required attributes for read and write operations (see Section 9.1).
- CLOSE—terminates the connection between a logical unit and a file or device (see Section 9.2).
- INQUIRE—questions the status of specified properties of a file or logical unit (see Section 9.3).
- REWIND—repositions an open file to the beginning of that file (see Section 9.4).
- BACKSPACE—repositions an open file to the beginning of the preceding record in that file (see Section 9.5).
- ENDFILE—writes an end-of-file record to a specified unit. When an input statement reads this record, an end-of-file condition results (see Section 9.6).
- DELETE—deletes a record from a file (see Section 9.7).
- UNLOCK—permits other programs to access a file that is locked by a previous READ statement (see Section 9.8).

9.1 OPEN Statement

The OPEN statement connects an existing file to a logical unit or creates a new file and connects it to a logical unit. In addition, it can specify file attributes that control file creation and subsequent processing. The OPEN statement takes the following form:

```
OPEN (par[,par]...)
```

par

Is a keyword specification taking one of the following forms:

```
keywd
keywd = value
```

keywd

Is a keyword, as described in the text that follows (see also Table 9–1).

value

Depends on the keyword (see Table 9–1).

Keywords can be divided into several categories based on function:

- Identifying the unit and 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
STATUS or TYPE	– file existence status at OPEN
DISPOSE	– file existence status after CLOSE

- Describing file processing:

ACCESS	– FORTRAN access method to be used
ORGANIZATION	– logical file structure
READONLY	– write protection

- Describing the records in a 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

- Describing file storage allocation when a file is created:

INITIALSIZE	– initial file allocation
EXTENDSIZE	– file allocation increment size

- Providing additional capability for direct access I/O:

ASSOCIATEVARIABLE	– the next record number value
MAXREC	– maximum direct access record number

- Providing improved performance or special capabilities; (these optional keywords 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
USEROPEN	– user program option to provide additional OPEN capability
SHARED	– other programs can simultaneously access the file
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 9–1 lists the values accepted for each keyword.

Table 9–1: OPEN Statement Keyword Values

Keyword	Values	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	c1	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	

Key to Values

asv—an integer variable
v—an integer scalar memory reference
e—a numeric expression
s—a statement label
dt—a data type, INTEGER or CHARACTER
dr—direction, ASCENDING or DESCENDING
c—a character scalar reference, numeric scalar memory reference, or numeric array name reference
c1—a character expression
e1—the first byte position of a key
e2—last byte position of a key
p—an external function

Table 9–1 (Cont.): OPEN Statement Keyword Values

Keyword	Values	Function	Default
EXTENDSIZE	e	File allocation increment	Volume or system default
FILE NAME	c	File-name specification	
FORM	'FORMATTED' 'UNFORMATTED'	Format type	Depends on ACCESS keyword
INITIALSIZE	e	File allocation	
IOSTAT	v	I/O status	
KEY	(e1:e2[:dt[:dr]],...)	Key field definitions	CHARACTER ASCENDING
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	Depends on record type and file organization

Key to Values

asv—an integer variable
v—an integer scalar memory reference
e—a numeric expression
s—a statement label
dt—a data type, INTEGER or CHARACTER
dr—direction, ASCENDING or DESCENDING
c—a character scalar reference, numeric scalar memory reference, or numeric array name reference
c1—a character expression
e1—the first byte position of a key
e2—last byte position of a key
p—an external function

Table 9–1 (Cont.): OPEN Statement Keyword Values

Keyword	Values	Function	Default
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	

Key to Values

asv—an integer variable
v—an integer scalar memory reference
e—a numeric expression
s—a statement label
dt—a data type, INTEGER or CHARACTER
dr—direction, ASCENDING or DESCENDING
c—a character scalar reference, numeric scalar memory reference, or numeric array name reference
c1—a character expression
e1—the first byte position of a key
e2—last byte position of a key
p—an external function

Specifying OPEN Statement Keywords

Keyword specifications can appear in any order. In most cases, they are optional. Default values apply in their absence. If the logical unit specifier is the first parameter in the list, the UNIT keyword is optional.

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 can contain trailing spaces but not leading or embedded spaces; for example:

```
CHARACTER*7 QUAL /' '/
    .
    .
    .
IF (exp) QUAL = '/DELETE'
OPEN (UNIT=1, STATUS='NEW', DISP='SUBMIT'//QUAL)
```

Examples

The first statement creates a new sequential formatted file on unit 1 with the default file name FOR001.DAT.

```
OPEN (UNIT=1, STATUS='NEW', ERR=100)
```

The next 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)
```

The next statement creates a file on magnetic tape with a large block size for efficient processing.

```
OPEN (UNIT=I, FILE='MTA0:MYDATA.DAT', BLOCKSIZE=8192,
1     STATUS='NEW', ERR=14, RECL=1024,
1     RECORDTYPE='FIXED')
```

The next statement opens the file created in the previous example for input.

```
OPEN (UNIT=I, FILE='MTA0:MYDATA.DAT', READONLY,
1     STATUS='OLD', RECL=1024, RECORDTYPE='FIXED',
1     BLOCKSIZE=8192)
```

The next 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')
```

The following sections provide specific information about OPEN statement 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.

9.1.1 ACCESS Keyword

The ACCESS parameter specifies whether a file opens for keyed, direct, or sequential access. It takes the following form:

```
ACCESS = acc
```

acc
Is a character expression having one of the following values:

- 'DIRECT'—by record number
- 'SEQUENTIAL'—by sequential access
- 'KEYED'—by a specified key
- 'APPEND'—sequentially, after the last record of the file

The default is 'SEQUENTIAL'.

9.1.2 ASSOCIATEVARIABLE Keyword

The ASSOCIATEVARIABLE parameter specifies the integer variable that updates 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 and is ignored for other access modes. It takes the following form:

```
ASSOCIATEVARIABLE = asv
```

asv
Is an integer variable. It cannot be a dummy argument to the routine in which the OPEN statement appears.

9.1.3 BLANK Keyword

The BLANK parameter specifies how empty spaces are treated in a file. It takes the following form:

```
BLANK = blnk
```

blnk
Is a character expression with one of the following values:

- 'NULL'—ignore all blanks in a numeric field (except if the field is all blank, in which case blanks are treated as zero)
- 'ZERO'—treat all blanks other than leading blanks as zeros

The default value is 'NULL'. However, if the /NOF77 qualifier is specified on the FORTRAN command line, the default is 'ZERO'.

9.1.4 BLOCKSIZE Keyword

The BLOCKSIZE parameter specifies the physical I/O transfer size for the file. It takes the following form:

```
BLOCKSIZE = bks
```

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 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 *VMS DCL Dictionary* for more information on setting process and system default multiblock counts if you do not specify a block size.

For relative 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 63 blocks. The default is the smallest value capable of holding a single record.

9.1.5 BUFFERCOUNT Keyword

The BUFFERCOUNT parameter specifies the number of buffers to be associated with the logical unit for multibuffered I/O. It takes the following form:

```
BUFFERCOUNT = bc
```

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.

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 *VMS DCL Dictionary* for information on setting process and system default buffer counts.

9.1.6 CARRIAGECONTROL Keyword

The CARRIAGECONTROL parameter determines the type of carriage control processing used when printing a file. It takes the following form:

```
CARRIAGECONTROL = cc
```

cc
Is a character expression taking one of the following values:

- 'FORTRAN'—normal FORTRAN interpretation of the first character
- 'LIST'—single spacing between records
- 'NONE'—no implied carriage control

The default for formatted files is 'FORTRAN'; for unformatted files, the default is 'NONE'.

9.1.7 DEFAULTFILE Keyword

The DEFAULTFILE parameter specifies a default file specification string. It takes the following form:

```
DEFAULTFILE = ce
```

ce

Is a character expression that contains a default file name specification string.

This keyword can 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 zeros.

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 default values for any one of the following file-specification components:

- Node
- Device
- Directory
- File name
- File type
- 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 *VMS Record Management Services Manual* for more information.

9.1.8 DISPOSE Keyword

The DISPOSE (or DISP) parameter determines the disposition of the file connected to a logical unit when the unit closes. It takes one of the following forms:

```
DISPOSE = dis
DISP = dis
```

dis

Is a character expression having one of the following values:

- 'KEEP' or 'SAVE'— retain the file after the unit closes
- 'DELETE'—delete the file after the unit closes
- 'PRINT'—submit the file to the system line printer spooler and retain it

- 'PRINT/DELETE'—submit the file to the system line printer spooler and then delete it
- 'SUBMIT'—submit the file to the batch job queue and retain it
- 'SUBMIT/DELETE'—submit the file to the batch job queue and then delete it

A read-only file cannot be deleted. A scratch file cannot be saved, printed, or submitted.

The default is 'KEEP' or 'SAVE'.

9.1.9 ERR Keyword

The ERR parameter identifies the executable statement that receives control when an error occurs. It takes the following form:

```
ERR = s
```

s
Is the label of an executable statement.

The ERR parameter 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.

9.1.10 EXTENDSIZE Keyword

The EXTENDSIZE parameter specifies the number of blocks by which to extend a disk file when additional storage space is allocated. It takes the following form:

```
EXTENDSIZE = es
```

es
Is a numeric expression.

If you do not specify EXTENDSIZE or if you specify zero, the system default for the device is used.

See Section 9.1.13 for a discussion about the relationship between the EXTENDSIZE keyword and the INITIALSIZE keyword.

9.1.11 FILE Keyword

The FILE parameter specifies the name of the file to be connected to the unit. It takes the following form:

```
FILE = fln
```

fln
Is a character scalar reference, numeric scalar memory reference, or numeric array name reference.

The name of a file can be any specification accepted by the operating system. (See the *VAX FORTRAN User Manual* for a description of 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.

9.1.12 FORM Keyword

The FORM parameter specifies whether the file being opened is read or written using formatted or unformatted READ or WRITE statements. It takes the following form:

```
FORM = ft
```

ft
Is a character expression taking one of the following values:

- 'FORMATTED'
- 'UNFORMATTED'

The default is 'FORMATTED' for sequential access files, and 'UNFORMATTED' for direct and keyed access files.

9.1.13 INITIALSIZE Keyword

The INITIALSIZE parameter specifies the number of blocks in the initial storage allocation for a disk file. It is contracted by the EXTENDSIZE parameter, which specifies the number of blocks by which a disk file is extended each time more space is needed for a file. The INITIALSIZE parameter takes the following form:

```
INITIALSIZE = insz
```

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.

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.

9.1.14 IOSTAT Keyword

The IOSTAT parameter specifies I/O status. It takes the following form:

```
IOSTAT = ios
```

ios
Is an integer scalar memory reference.

This parameter causes ios to be defined as follows:

- Zero—if no error condition exists
- A positive integer—if an error condition exists

VAX FORTRAN I/O status values are described in the *VAX FORTRAN User Manual*. 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, you can use the IOSTAT parameter in subsequent I/O statements to perform a similar function (see Section 7.1.1.9).

9.1.15 KEY Keyword

The KEY parameter defines the access keys for records in an indexed file. It takes the following form:

```
KEY = (kspec[,kspec]...)
```

kspec
Takes the following form:

```
e1:e2[:dt[:dr]]
```

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: INTEGER or CHARACTER.

dr
Is the direction of the key: ASCENDING or DESCENDING.

CHARACTER and ASCENDING are the default values.

The key starts at position e1 in a record and has a length of $e2 - e1 + 1$. The values of e1 and e2 must be such that the following calculations are true:

```
1 .LE. (e1) .AND. (e1) .LE. (e2) .AND. (e2) .LE. record-length
1 .LE. (e2-e1+1) .AND. (e2-e1+1) .LE. 255
```

If the key type is INTEGER, the key length must be either 2 or 4.

Defining Primary and Alternate Keys

You must define at least one key in an indexed file. This primary key is the default key. It usually has a unique value for each record.

You can choose to define alternate keys. RMS allows up to 254 alternate keys. However, individual OPEN statements only allow up to 85 key definitions, a number that is further reduced when multiple OPEN statements appear together in a program unit.

If a file requires more keys than the OPEN statement limit, you must create it from another language or with the File Definition Language (FDL). For information on FDL, see the *VMS Record Management Services Manual*.

Specifying and Referencing Keys

You must specify the KEY parameter when creating an indexed file. However, you do not have to respecify it when opening an existing file because key attributes are permanent aspects of the file. These attributes include key definitions and reference numbers for subsequent I/O operations. If you do choose to specify the KEY parameter for an existing file, your specification must be identical to the established key attributes.

Subsequent I/O operations use a reference number, called the key-of-reference number, to identify a particular key. You do not specify this number; it is determined by the key's position in the specification list: the primary key is key-of-reference number 0; the first alternate key is key-of-reference number 1, and so forth.

9.1.16 MAXREC Keyword

The MAXREC parameter applies only to direct access files. It specifies the maximum number of records permitted in a direct access file. The MAXREC parameter takes the following form:

```
MAXREC = mr
```

mr
Is a numeric expression.

The default is an unlimited number of records.

9.1.17 NAME Keyword

NAME is a nonstandard synonym for FILE. See Section 9.1.11.

9.1.18 NOSPANBLOCKS Keyword

The NOSPANBLOCKS parameter specifies that records are not to cross disk block boundaries. It takes the following form:

```
NOSPANBLOCKS
```

When you specify this parameter, an error occurs if any record exceeds the size of a physical block.

9.1.19 ORGANIZATION Keyword

The ORGANIZATION parameter specifies the internal organization of the file. It takes the following form:

```
ORGANIZATION = org
```

org
Is a character expression with one of the following values:

- 'SEQUENTIAL'
- 'RELATIVE'
- 'INDEXED'

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.

9.1.20 READONLY Keyword

The READONLY parameter specifies that an existing file can be read and prohibits writing to that file. It takes the following form:

```
READONLY
```

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 Manual* for information on file sharing.

9.1.21 RECL Keyword

The RECL parameter specifies the length of logical records in a file. It takes the following form:

```
RECL = rl
```

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 expressed in units of bytes or longwords, depending on the record's format. Formatted records use byte units and unformatted records use longword units (which are equal to 4 bytes). Table 9–2 lists the maximum values that can be specified for rl, based on file organization and record format.

Table 9–2: 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.

RECL is mandatory when opening files with fixed-length records or relative or indexed organization; it is optional when opening all other types. Default values for optional cases depend on the value of the RECORDTYPE parameter. Table 9–3 lists the RECL default values.

Table 9–3: Record Size (RECL) Default Values

RECORDTYPE value	RECL value
'FIXED'	none; value must be explicitly specified
'SEGMENTED'	2048
All other types	133

The interpretation and effect of the logical record length varies as follows:

- 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 your program attempts to write to an existing file a record that is longer than the logical record length, an error occurs.
- 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.

9.1.22 RECORDSIZE Keyword

RECORDSIZE is a nonstandard synonym for RECL; refer to Section 9.1.21 for more information.

9.1.23 RECORDTYPE Keyword

The RECORDTYPE parameter specifies whether the file has fixed-length records, variable-length records, segmented records, or stream-type variable-length records. It takes the following form:

```
RECORDTYPE = typ
```

typ
Is a character expression with one of the following values:

- 'FIXED'
- 'VARIABLE'
- 'SEGMENTED'
- 'STREAM'
- 'STREAM_CR'
- 'STREAM_LF'

When you create a file, default record types are as follows:

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. Using segmented records allows a FORTRAN logical record to span several physical 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— except for 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 in a formatted file and zeros in 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.

9.1.24 SHARED Keyword

The SHARED parameter specifies that the file can be opened for shared access by more than one program executing simultaneously. It takes the following form:

```
SHARED
```

For information on file sharing, see the *VAX FORTRAN User Manual*.

9.1.25 STATUS Keyword

The STATUS parameter specifies the status of the file that you wish to open. It takes the following form:

```
STATUS = sta
```

sta

Is a character expression with one of the following values:

- 'OLD'—the file must already exist.
- 'NEW'—a new file is created.
- 'SCRATCH'—a new file is created and is deleted when it closes.

- 'UNKNOWN'—the processor first tries 'OLD'; if the file is not found, the processor uses 'NEW', thereby creating a new file.

The default is 'UNKNOWN'. However, if you specify the /NOF77 qualifier on the FORTRAN command line, the default value specified by the compiler 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.

9.1.26 TYPE Keyword

TYPE is a nonstandard synonym for STATUS. See Section 9.1.25.

9.1.27 UNIT Keyword

The UNIT parameter specifies the logical unit that connects to the file. It takes the following form:

```
[UNIT=] u
```

u
Is a numeric expression.

The unit specification must appear in the parameter list unless 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.

See the *VAX FORTRAN User Manual* for additional information about logical unit numbers.

9.1.28 USEROPEN Keyword

The USEROPEN parameter specifies a user-written external function that controls the opening of the file. It takes the following form:

```
USEROPEN = procedure-name
```

procedure-name
Is the symbolic name of the USEROPEN procedure.

The procedure name must be declared EXTERNAL.

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 Manual* for more information on USEROPEN.

9.2 CLOSE Statement

The CLOSE statement disconnects a file from a unit. It takes the following form:

```
CLOSE ([UNIT=]u[, {STATUS  } =p][,ERR=s][,IOSTAT=ios])
                  {DISPOSE }
                  {DISP    }
```

u
Is a logical unit number.

p
Is a character expression that determines the disposition of the file. It can be any one of the following values:

- 'SAVE' or 'KEEP'— retain the file after the unit closes
- 'DELETE'– delete the file
- 'PRINT'– submit the file to the line printer spooler and retain it
- 'PRINT/DELETE'—submit the file to the line printer spooler and then delete it
- 'SUBMIT'– submit the file to the batch job queue and retain it
- 'SUBMIT/DELETE'—submit the file to the batch job queue and then delete it

The default is 'DELETE' for scratch files. For all other files, the default is 'KEEP'.

s
Is the label of an executable statement.

ios
Is an integer scalar memory reference.

Syntax Rules and Behavior

CLOSE statement parameters can occur in any order. The UNIT keyword is optional only if the unit specifier is the first parameter in the list.

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, the following statement closes the file on unit 1 and submits it for printing:

```
CLOSE (UNIT=1, STATUS='PRINT')
```

The next statement closes the file on unit J and deletes it:

```
CLOSE (UNIT=J, STATUS='DELETE', ERR=99)
```

9.3 INQUIRE Statement

The INQUIRE statement "asks" about specified properties of a file or of a logical unit on which a file might be opened. It takes two forms, one for inquiring by file and the other for inquiring by unit:

Inquiring by File

```
INQUIRE (FILE=fi[,DEFAULTFILE=dfi...],flist)
```

Inquiring by Unit

```
INQUIRE ([UNIT=]u,flist)
```

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 the following sections.

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.

Syntax Rules and Behavior

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 open with both FILE and DEFAULTFILE 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 executes.

9.3.1 ACCESS Specifier

The ACCESS specifier takes the following form:

```
ACCESS = acc
```

acc
Is a character scalar memory reference that is assigned one of the following values:

- SEQUENTIAL—if the file is open for sequential access
- DIRECT—if the file is open for direct access
- KEYED—if the file is open for keyed access
- UNKNOWN—if no connection exists

9.3.2 BLANK Specifier

The BLANK specifier takes the following form:

```
BLANK = blk
```

blk
Is a character scalar memory reference that is assigned one of the following values:

- NULL—if null blank control is in effect for a file open for formatted I/O
- ZERO—if zero blank control is in effect
- UNKNOWN—if no connection exists or if the connection is not for formatted I/O

9.3.3 CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier takes the following form:

```
CARRIAGECONTROL = cc
```

cc
Is a character scalar memory reference that is assigned one of the following values:

- FORTRAN—if the file has FORTRAN carriage control
- LIST—if the file has implied carriage control
- NONE—if the file has no carriage control attribute
- UNKNOWN—if no other values apply

9.3.4 DIRECT Specifier

The DIRECT specifier takes the following form:

```
DIRECT = dir
```

dir
Is a character scalar memory reference that is assigned one of the following values:

- YES—if direct access is allowed for the file
- NO—if direct access is not allowed
- UNKNOWN—if the processor cannot determine whether direct access is allowed

9.3.5 ERR Specifier

The ERR specifier takes the following form:

```
ERR = s
```

s
Is the label of an executable statement.

The ERR specifier 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.

9.3.6 EXIST Specifier

The EXIST specifier takes the following form:

```
EXIST = ex
```

ex
Is a logical scalar memory reference that is assigned one of the following values:

- .TRUE.—if the specified file or unit exists
- .FALSE.—if the specified file or unit does not exist or the file cannot be opened even though it exists

9.3.7 FORM Specifier

The FORM specifier takes the following form:

```
FORM = fm
```

fm
Is a character scalar memory reference that is assigned one of the following values:

- FORMATTED—if the file is open for formatted I/O
- UNFORMATTED—if the file is open for unformatted I/O
- UNKNOWN—if no connection exists

9.3.8 FORMATTED Specifier

The FORMATTED specifier takes the following form:

```
FORMATTED = fmd
```

fmd
Is a character scalar memory reference that is assigned one of the following values:

- YES—if formatted is an allowed form for the file
- NO—if formatted is not an allowed form
- UNKNOWN—if the processor is unable to determine whether formatted is an allowed form

9.3.9 IOSTAT Specifier

The IOSTAT specifier takes the following form:

```
IOSTAT = ios
```

ios

Is an integer scalar memory reference.

The IOSTAT specifier is a control specifier rather than a property specifier. Ios can be assigned one of the following values:

- If an error occurs during execution of the INQUIRE statement, it takes a processor-dependent positive integer value.
- If no error occurs, it takes a ZERO value.

9.3.10 KEYED Specifier

The KEYED specifier takes the following form:

```
KEYED = kyd
```

kyd

Is a character scalar memory reference that is assigned one of the following values:

- YES—if keyed access is allowed for the file (The file must be indexed.)
- NO—if keyed access is not allowed
- UNKNOWN—if the processor is unable to determine whether keyed access is allowed

9.3.11 NAME Specifier

The NAME specifier takes the following form:

```
NAME = nme
```

nme

Is a character scalar memory reference that is assigned the name of the file being inquired about. If the file does not have a name, nme is undefined.

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 with the FILE keyword in an OPEN statement.

NOTE

The FILE and NAME keywords are synonyms when used with the OPEN statement, but not when used with the INQUIRE statement.

9.3.12 NAMED Specifier

The NAMED specifier takes the following form:

```
NAMED = nmd
```

nmd
Is a logical scalar memory reference that is assigned one of the following values:

- .TRUE.—if the specified file has a name
- .FALSE.—if it does not have a name

9.3.13 NEXTREC Specifier

The NEXTREC specifier takes the following form:

```
NEXTREC = nr
```

nr
Is an integer scalar memory reference that is assigned one of the following values:

- If a record was previously read or written on the specified unit, the value of nr is one more than the number of that record.
- If no records have been read or written, the value of nr is one.
- If the file is not opened for direct access or if the position is indeterminate because of an error condition, nr is zero.

9.3.14 NUMBER Specifier

The NUMBER specifier takes the following form:

```
NUMBER = num
```

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.

9.3.15 OPENED Specifier

The OPENED specifier takes the following form:

```
OPENED = od
```

od
Is a logical scalar memory reference that is assigned one of the following values:

- .TRUE.—if the specified file is open on a unit or if the specified unit is open
- .FALSE.—if the specified file or unit is not open

9.3.16 ORGANIZATION Specifier

The ORGANIZATION specifier takes the following form:

```
ORGANIZATION= org
```

org
Is a character scalar memory reference that is assigned one of the following values:

- SEQUENTIAL—if the file is a sequential file
- RELATIVE—if the file is a relative file
- INDEXED—if the file is an indexed file
- UNKNOWN—if the processor is unable to determine the file's organization

9.3.17 RECL Specifier

The RECL specifier takes the following form:

```
RECL = rcl
```

rcl

Is an integer scalar memory reference taking one of the following values:

- If the file or unit is open, rcl is the maximum record length allowed in the file.
- If the file is not open, rcl is the maximum record length allowed in the file; or, if the maximum record length is 0, rcl is the length of the longest record in the file.

  If inquiring about a file that has no maximum record size, see Section 9.1.21.
- If the file is segmented, rcl is the longest segment length in the file.
- If a specified file does not exist, rcl is zero.

The rcl value is expressed in longwords if a file is (or has been) opened for unformatted I/O; and in bytes in all other circumstances.

9.3.18 RECORDTYPE Specifier

The RECORDTYPE specifier takes the following form:

```
RECORDTYPE = rtype
```

rtype

Is a character scalar memory reference that is assigned one of the following values:

- FIXED—if the file is open for fixed-length records
- VARIABLE—if the file has variable-length records
- SEGMENTED—if the file is open for unformatted sequential I/O using segmented records
- STREAM—if the file's records are terminated with a carriage-return and line-feed
- STREAM_CR—if the file's records are terminated only with a carriage-return

- STREAM_LF—if the file's records are terminated only with line-feed
- UNKNOWN—if the processor cannot determine the record type

9.3.19 SEQUENTIAL Specifier

The SEQUENTIAL specifier takes the following form:

```
SEQUENTIAL = seq
```

seq
Is a character scalar memory reference that is assigned one of the following values:

- YES—if sequential access is allowed for the specified file
- NO—if sequential access is not allowed
- UNKNOWN—if the processor cannot determine whether sequential access is allowed

9.3.20 UNFORMATTED Specifier

The UNFORMATTED specifier takes the following form:

```
UNFORMATTED = unf
```

unf
Is a character scalar memory reference that is assigned one of the following values:

- YES—if unformatted is an allowed form for the file
- NO—if unformatted is not an allowed form for the file
- UNKNOWN—if the processor is unable to determine whether unformatted is an allowed form for the file

9.4 REWIND Statement

The REWIND statement repositions a sequential file currently open for sequential or append access to the beginning of the file. It takes either one of the following forms:

```
REWIND ([UNIT=]u[,ERR=s][,IOSTAT=ios])

REWIND u
```

u
Is a logical unit number.

s
Is the label of the executable statement that receives control 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.

Syntax Rules and Behavior

The unit number must refer to a file on disk or magnetic tape. For example, the following statement repositions logical unit 3 to the beginning of the currently open file:

```
REWIND 3
```

This statement repositions logical unit 3 to the beginning of the currently open file.

A REWIND statement should not be issued for a file that is open for direct or keyed access.

9.5 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, the preceding record is available for processing. The BACKSPACE statement takes one of the following forms:

```
BACKSPACE ([UNIT=]u[,ERR=s][,IOSTAT=ios])

BACKSPACE u
```

u
Is a logical unit number.

s
Is the label of the executable statement that receives control 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.

Syntax Rules and Behavior

The unit number must refer to an open file on disk or magnetic tape. For example, the following statement repositions the open file on logical unit 4 to the beginning of the preceding record:

```
BACKSPACE 4
```

A BACKSPACE statement should not be issued 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.

9.6 ENDFILE Statement

The ENDFILE statement writes an end-file record to the specified unit. It takes one of the following forms:

```
ENDFILE ([UNIT=]u[,ERR=s][,IOSTAT=ios])

ENDFILE u
```

u
Is a logical unit number.

s
Is the label of the executable statement that receives control 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.

Syntax Rules and Behavior

If the unit specified in the ENDFILE statement is not open, the default file is opened for unformatted output.

An end-file record can be written only to files with sequential organization that are accessed as formatted-sequential or unformatted-segmented sequential files. For example, the following statement writes an end-file record to the logical unit 2:

```
ENDFILE 2
```

An ENDFILE statement must not be issued for a file that is open for direct or keyed 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).

9.7 DELETE Statement

The DELETE statement deletes records from relative and indexed files. It takes one of the following 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])
```

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 that receives control 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.

Syntax Rules and Behavior

The form of the DELETE statement for indexed files is a current-record delete. This form deletes the current record, which is the last record that is accessed by a READ statement on the specified logical unit.

The forms of the DELETE statement with relative files are direct access deletes. These forms delete the record specified by the number r.

The DELETE statement logically removes the appropriate record from the specified file by locating the record and marking 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.

Following a direct access delete, any associated variable is set to the next record number.

In the following example, the fifth record in the file connected to logical unit 10 is deleted from the file:

```
DELETE (10,REC=5)
```

In the next example, the current record is deleted from the file connected to logical unit 11:

```
DELETE (11)
```

9.8 UNLOCK Statement

The UNLOCK statement frees a record in an indexed, relative, or sequential file that was locked on a specified logical unit by a previous READ. It performs no other I/O operation. The UNLOCK statement takes one of the following forms:

```
UNLOCK ([UNIT=]u[,ERR=s][,IOSTAT=ios])

UNLOCK u
```

u
Is the number of a logical unit.

s
Is the label of the executable statement that receives control 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.

If no record is locked, the operation has no effect.

Chapter 10

Compiler Directives

Compiler directives tell the compiler to perform certain tasks when it compiles a source program unit. They are preceded by special tags that identify them to the compiler.

VAX FORTRAN provides two categories of compiler directives: one category, *parallel directives*, supports directed decomposition for parallel processing. Parallel directives are preceded by the CPAR$ tag. The other category, *general directives*, provides several general-purpose functions. General directives are preceded by the CDEC$ tag.

10.1 Compiler Directive Syntax Rules

The following general syntax rules apply to all compiler directives. They must be precisely followed to properly compile your program and obtain meaningful results:

- The tag (CPAR$ or CDEC$) must appear in columns 1 through 5.
- Column 6 must be a blank or tab.
- From column 7 on, blanks are insignificant. Thus, the directive can be positioned anywhere on the line after column 6.
- Continuation lines cannot appear in compiler directives.
- If a blank common block is used in a compiler directive, it must be specified as two slashes (/ /).

See the individual sections in this chapter for listings of rules that pertain only to specific categories or individual directives.

10.2 Parallel Directives

Parallel directives invoke parallel processing of indexed DO-loops, synchronize execution of critical regions within the loops, and define the sharability of common blocks and symbols in parallel applications.

The /PARALLEL qualifier enables parallel directives. See the *VAX FORTRAN User Manual* for details about this qualifier.

Parallel directives take the following form:

```
column 1
↓
CPAR$ directive
```

directive
Is any one of the following values:

- CONTEXT_SHARED—specifies shared memory locations for symbols declared in routines that contain parallel DO-loops.
- CONTEXT_SHARED_ALL—reinforces the context-shared default of symbols in routines compiled with the /PARALLEL qualifier.
- DO_PARALLEL—enables parallel processing of indexed DO-loops.
- LOCKON and LOCKOFF—forces processes to sequentially execute a region of code.
- SHARED—specifies shared common blocks for parallel processing.
- SHARED_ALL—reinforces the shared default of common blocks in routines compiled with the /PARALLEL qualifier.
- PRIVATE—specifies unique (private) common blocks or symbols for each process.
- PRIVATE_ALL—forces all common blocks and symbols to have PRIVATE defaults in routines compiled with the /PARALLEL qualifier.

Section 10.2.9 provides examples of parallel directives appropriately used in a program unit.

10.2.1 CPAR$ CONTEXT_SHARED

The CONTEXT_SHARED directive specifies that the same memory location will be used for a symbol declared in a routine. The symbol uses the same memory location in any parallel DO-loops contained within the routine.

If a routine has several concurrent invocations (because it is called from within a parallel DO-loop), each invocation uses different memory locations for its variables that are declared CONTEXT_SHARED.

The CONTEXT_SHARED directive takes the following form:

```
CPAR$ CONTEXT_SHARED syname [,syname]...
```

syname
Is the name of a variable, array, or record declared within the routine.

Syntax Rules

In addition to the general compiler-directive syntax rules listed in Section 10.1, CONTEXT_SHARED has the following specific rules:

- CONTEXT_SHARED can appear anywhere within declaration statements in the routine.
- Arrays cannot be dimensioned within CONTEXT_SHARED.
- Symbols listed in CONTEXT_SHARED cannot also be listed in a PRIVATE directive.
- Dummy arguments cannot be declared CONTEXT_SHARED.

10.2.2 CPAR$ CONTEXT_SHARED_ALL

The CONTEXT_SHARED_ALL directive forces all symbols that are declared within a routine to default to CONTEXT_SHARED. It takes the following form:

```
CPAR$ CONTEXT_SHARED_ALL
```

This directive affects only default behavior. Individual symbols can still be declared PRIVATE.

All general compiler-directive syntax rules apply to the CONTEXT_SHARED directive (see Section 10.1).

10.2.3 CPAR$ DO_PARALLEL

The DO_PARALLEL directive tells the compiler to run an indexed DO-loop in parallel. It takes the following form:

```
CPAR$ DO_PARALLEL [count]
```

count

Is a numeric expression specifying the number of iterations in each set distributed to processes running the DO-loop. Any remaining iterations run in the last process. For example, if the total number of iterations is 1330, count is 100, and two processes are running the DO-loop, then each process is distributed sets of 100 iterations at a time. The last set contains the last 30 iterations.

The numeric expression must evaluate to a positive, non-zero integer. If necessary, the process converts it to an integer by truncation. For example, 50.56 is acceptable and is converted to 50. However, 0.20 is not acceptable because it is converted to zero.

Syntax Rules

In addition to the general compiler-directive syntax rules listed in Section 10.1, DO_PARALLEL and its corresponding DO-loop have the following specific rules:

- The indexed DO statement must be the next executable statement following the DO_PARALLEL directive. Only comments can appear between the directive and its corresponding DO-loop.
- The parallel DO-loop control variable must be a private integer. If it is not specified PRIVATE, the compiler gives an informational message and changes the control variable's default to PRIVATE. If the control variable is not an integer, the compiler gives an error.
- A parallel DO-loop cannot contain the following:
  - I/O statements
  - PAUSE, RETURN, or STOP statements
  - Calls to system services or run-time library routines that modify the context of the process
- A parallel DO-loop must not have a branch into or out of its body.

When calculations in parallel DO-loops require serial (sequential) execution to achieve correct results, you must guard those calculations (called *critical regions*) against parallel execution. This situation occurs when calculations modify shared values that are needed in successive iterations of the parallel DO-loop or elsewhere in the routine.

The following section describes the LOCKON and LOCKOFF directives, which guard critical regions. See also the *VAX FORTRAN User Manual* for additional information.

10.2.4 CPAR$ LOCKON, CPAR$ LOCKOFF

The LOCKON and LOCKOFF directives enclose a region of executable code within a parallel DO-loop. They effectually force processes to execute this critical region one at a time by excluding all other processes from the region while one process is executing it.

A process executing the critical region "has the lock" while other processes that want to execute it must wait until it is free. The lock becomes free when the process in the critical region executes the associated LOCKOFF directive.

The LOCKON and LOCKOFF directives take the following forms:

```
CPAR$ LOCKON lck-var
      (critical region)
CPAR$ LOCKOFF lck-var
```

lck-var

Is a LOGICAL*4 variable that can be any of the following items:

- Scalar variable
- Dummy argument
- Record field
- Array element

The lock variable (lck-var) must be in a shared common block or a variable declared as CONTEXT_SHARED in the routine containing a parallel DO-loop.

The lock variable is considered locked when it has a value of .TRUE. and free when it has a value of .FALSE.

Syntax Rules

In addition to the general compiler-directive syntax rules listed in Section 10.1, LOCKON and LOCKOFF have the following specific rules:

- All LOCKON directives require an associated LOCKOFF directive that executes in the DO-loop.
- Statements in critical regions cannot transfer control outside the region.
- Lock variables can only have a LOGICAL*4 data type. Other data types cause a compilation error.
- Lock variables must be shared, either by specification or default. If they are declared private, a compilation error results.

10.2.5 CPAR$ PRIVATE

The PRIVATE directive specifies the common blocks and symbols that must be unique for each process that runs the parallel DO-loop. It takes the following form:

```
CPAR$ PRIVATE name [,name]...
```

name

Is the name of a symbol or a common block (preceded and followed by a slash). Common block names cannot exceed 26 characters; they can be blank. Symbols can be any variable name (scalar, array, or record) that is declared within the routine.

Syntax Rules

In addition to the general compiler-directive syntax rules listed in Section 10.1, PRIVATE has the following specific rules:

- PRIVATE directives can appear anywhere within declaration statements in the routine.
- Arrays cannot be dimensioned within this directive.
- Elements contained in a common block cannot be declared PRIVATE.
- Common blocks that are declared PRIVATE cannot also be declared SHARED.
- Symbols declared PRIVATE cannot also be declared CONTEXT_SHARED.

- Symbols can only be declared PRIVATE in routines containing a parallel DO-loop.
- Dummy arguments cannot be declared PRIVATE.
- SAVE statements cannot refer to PRIVATE symbols or common blocks.

10.2.6 CPAR$ PRIVATE_ALL

The PRIVATE_ALL directive forces all symbols and common blocks to default to PRIVATE in each process that runs the parallel DO-loop. It takes the following form:

```
CPAR$ PRIVATE_ALL
```

The PRIVATE_ALL directive changes only default behavior. Individual common blocks can still be declared SHARED and individual symbols can still be declared CONTEXT_SHARED.

All general compiler-directive syntax rules apply to the PRIVATE_ALL directive (see Section 10.1).

10.2.7 CPAR$ SHARED

The SHARED directive specifies the common blocks that must be shared between processes running a parallel DO-loop. It takes the following form:

```
CPAR$ SHARED /[cb]/[,/[cb]/]...
```

cb
Is the name of a common block. The name cannot exceed 26 characters; it can be blank.

Syntax Rules

In addition to the general compiler-directive syntax rules listed in Section 10.1, SHARED has the following specific rules:

- SHARED directives can appear anywhere within declaration statements in the routine.
- Common blocks that are declared SHARED cannot also be declared PRIVATE.

10.2.8 CPAR$ SHARED_ALL

The SHARED_ALL directive forces all common blocks to have SHARED defaults among processes that use the parallel DO-loop. It takes the following form:

```
CPAR$ SHARED_ALL
```

This directive affects only default behavior. Individual common blocks can still be declared PRIVATE.

All compiler-directive syntax rules apply to the SHARED_ALL directive (see Section 10.1).

10.2.9 Parallel Directive Examples

The following examples demonstrate valid applications of parallel directives.

In the first example, assignment to the context-shared variables SUMA and SUMB must be guarded from multiple processes attempting to write to the variable at the same time. The LOCKON and LOCKOFF directives accomplish this task by ensuring scalar execution of the statements.

```
CPAR$ PRIVATE I
CPAR$ SHARED /COM1/
CPAR$ CONTEXT_SHARED SUMA, SUMB

        COMMON/COM1/A, B
        INTEGER A(1000), B(1000), SUMA, SUMB
        LOGICAL *4 LCK_VAR

CPAR$ DO_PARALLEL
        DO I = 1, 1000
        CALL CALCULATE (I)

CPAR$ LOCKON LCK_VAR            ! Guard against multiple processes
        SUMA = SUMA + A(I)      ! writing to the context-shared
        SUMB = SUMB + B(I)      ! variable at the same time.
CPAR$ LOCKOFF LCK_VAR
        ENDO

        PRINT*, 'SUM A=', SUMA
        PRINT*, 'SUM B=', SUMB

        END
```

The second example uses valid SHARED, CONTEXT_SHARED, and PRIVATE directives.

```
        INTEGER A(1000), B(1000)
        COMMON/COM1/B

        PARAMETER (N = 1000)

CPAR$ SHARED_ALL                ! Reinforces the SHARED default for
                                ! common blocks.
CPAR$ CONTEXT_SHARED_ALL        ! Reinforces the CONTEXT_SHARED default
                                ! for local symbols.

CPAR$ PRIVATE I                 ! Loop control must be private.

CPAR$ DO_PARALLEL
        DO 10 I = 1, N
        .
        .
        .
        A(I) = A(I) + I
        B(I) = A(I)
        .
        .
        .
10      CONTINUE

        CALL SUBR(A, N)

        WRITE (5,*) A, B

        END
C
        SUBROUTINE SUBR(A, N)

        INTEGER A(N), B(1000)
        COMMON/COM1/B

CPAR$ SHARED /COM1/             ! The common block must be SHARED.
CPAR$ PRIVATE_ALL               ! Make the default PRIVATE.

CPAR$ DO_PARALLEL N/2           ! Distributes half of the iterations to
        DO J = 1, N             ! each of two processors.
        .
        .
        .
        A(J) = B(J) + J
        .
        .
        .
        ENDO

        RETURN
        END
```

10.3 General Directives

General directives label or modify certain entities. No qualifier is needed on the FORTRAN command line to enable general directives.

General directives take the following form:

```
column 1
↓
CDEC$ directive
```

directive
Is any one of the following values:

- IDENT—provides identification of an object module.
- PSECT—modifies certain attributes of a common block.
- TITLE, SUBTITLE—provides a listing header.

10.3.1 CDEC$ IDENT

The IDENT directive specifies a string that can be used to identify an object module. The compiler places the string in the identification field of an object module when it generates the module for each source program unit. The IDENT directive takes the following form:

```
CDEC$ IDENT string
```

string
Is a group of up to 31 printable characters delimited by apostrophes.

Syntax Rules

In addition to the general compiler-directive syntax rules listed in Section 10.1, IDENT has the following specific rules:

- Only the first IDENT directive is effective—the compiler ignores any additional IDENT directives in a program unit.
- IDENT has no effect when you specify /NOOBJECT on the FORTRAN command line.

10.3.2 CDEC$ PSECT

The PSECT directive modifies several attributes of a common block. It takes the following form:

```
CDEC$ PSECT /common-name/ attr [,attr]...
```

common-name
Is the name of the common block, preceded and followed by a slash.

attr
Is one of the following attributes:

- LCL—local scope; opposite to GBL and cannot appear with it
- GBL—global scope
- [NO]WRT—writability or no-writability
- [NO]SHR—shareability or no-shareability
- ALIGN=val—alignment for the common block; val must be a constant ranging from 0 thru 9

Syntax Rules

In addition to the general compiler-directive syntax rules listed in Section 10.1, PSECT has the following specific rules:

- If one program unit changes one or more attributes, all other units that reference the common block must also change those attributes in the same way.
- Default attributes apply if you do not modify them with a PSECT directive. Table 10–1 lists the default attributes of common blocks and how they can be modified by PSECT.

Table 10–1: Common Block Default Attributes and PSECT Modification

Common Block Default Attributes	PSECT Modification
Relocatability	none
Overlaid	none
Global Scope	Global or local scope
No executability	none
Writability	Writability or no-writability
Readability	none
Position independence	none
Shareability	Shareability or no-shareability
No protection	none
LONGWORD alignment (2) (FORTRAN default)	0 thru 9

Global or *local scope* is significant for an image that has more than one cluster. The attribute determines whether program sections with the same name but from different modules in different clusters are finally placed in separate clusters (local scope) or in the same cluster (global scope).

[No]writability determines whether the contents of a common block can be modified during program execution.

[No]shareability determines whether the contents of a common block can be shared by more than one process.

ALIGN = val aligns the common block. The specified number is interpreted as a power of 2. The value of the expression is the alignment in bytes: a value of 0 specifies BYTE alignment; a value of 9 specifies page alignment.

Refer to the *VMS Linker Utility Manual* for detailed information about default attributes of common blocks.

10.3.3 CDEC$ TITLE, CDEC$ SUBTITLE

The TITLE directive specifies a string and places it in the title field of a listing header. Similarly, SUBTITLE places a specified string in the subtitle field of a listing header.

These directives take the following forms:

```
CDEC$ TITLE string
CDEC$ SUBTITLE string
```

string
Is a group of up to 31 printable characters delimited by apostrophes.

Syntax Rules

In addition to the general compiler-directive syntax rules listed in Section 10.1, TITLE and SUBTITLE have the following specific rules:

- To enable TITLE and SUBTITLE directives, you must specify /LIST on the FORTRAN command line.
- When TITLE or SUBTITLE appear on a page of a listing file, the specified string appears in the listing header of the following page.
- If two or more of either directives appear on a page, the last directive is the one in effect for the following page.
- If either directive does not specify a string, no change occurs in the listing file header.

Appendix A

Additional Language Features

To facilitate compatibility with other versions of FORTRAN, VAX FORTRAN provides the following additional language features:

- ENCODE, DECODE, DEFINE FILE, and FIND statements
- Alternative syntax for the PARAMETER statement and octal constants
- A /NOF77 interpretation of the EXTERNAL statement

These language features are particularly useful in transporting older FORTRAN programs to a VAX system. However, you should avoid using them in new programs on VAX systems and in new programs for which portability to other FORTRAN–77 implementations is important.

A.1 The ENCODE and DECODE Statements

The ENCODE and DECODE statements transfer data between variables or arrays in internal storage. The ENCODE statement translates data from internal (binary) form to character form. Inversely, the DECODE statement translates data from character to internal form. These statements are comparable to using internal files in formatted sequential WRITE and READ statements, respectively.

The ENCODE and DECODE statements take the following forms:

```
ENCODE (c,f,b[,IOSTAT=ios][,ERR=s]) [list]
DECODE (c,f,b[,IOSTAT=ios][,ERR=s]) [list]
```

c
Is an integer expression. In the ENCODE statement, c is the number of characters (in 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. An error occurs if more than one record is specified.

b
Is a scalar or array name reference. If b is an array name reference, its elements are processed in the order of subscript progression.

In the ENCODE statement, b receives the characters after translation to external form. If less than c characters are received, the remaining character positions are filled with blank characters. 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 zero if no error occurs.

s
Is the label of an executable statement.

list
Is an I/O list.

In the ENCODE statement, the list contains the data to be translated to character form. In the DECODE statement, the list receives the data after translation to internal form.

The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

Syntax Rules and Behavior

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.

Examples

In the following example, the DECODE statement translates the 12 characters in A to integer form (as specified by FORMAT 100):

```
     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 12 characters are stored in array K:

```
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:

```
B = '901256781234'
```

A.2 DEFINE FILE Statement

The DEFINE FILE statement establishes the size and structure of files with relative organization and associates them with a logical unit number. The DEFINE FILE statement is comparable to the OPEN statement (in situations where you can use the OPEN statement, it is the preferable mechanism for creating and opening files).

The DEFINE FILE statement takes the following form:

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

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.

Syntax Rules and Behavior

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 statement, a new relative organization file is created. If it is a READ or FIND statement, an existing file is opened, unless the specified file does not exist. If a file does not exist, an error occurs.

The DEFINE FILE statement 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.

Example

In the following example, the DEFINE FILE statement specifies that the 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.

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

A.3 FIND Statement

The FIND statement positions a direct access file at a particular record and sets the associated variable of the file to that record number. It is comparable to a direct access READ statement with no I/O list, and can open an existing file. 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 takes one of the following forms:

```
FIND (u'r[,ERR=s][,IOSTAT=ios])

FIND ([UNIT=]u,REC=r[,ERR=s][,IOSTAT=ios])
```

u
Is a logical unit number. It must refer to a relative organization file.

r
Is the direct access record number. It cannot be less than one or greater than the number of records defined for the file.

s
Is the label of the executable statement that receives control 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.

Examples

In the first example, the FIND statement positions logical unit 1 at the first record in the file. The file's associated variable is set to one:

```
FIND (1'1)
```

In the second example, the FIND statement positions the file at the record identified by the content of INDX. The file's associated variable is set to the value of INDX:

```
FIND (4'INDX)
```

A.4 PARAMETER Statement

The PARAMETER statement discussed here is similar to the one discussed in Section 4.11; they both assign a symbolic name to a constant. However, this PARAMETER statement differs from the other one in the following two 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.

This PARAMETER statement takes the following form:

```
PARAMETER  p=c [,p=c]...
```

p
Is a symbolic name.

c
Is a constant, the symbolic name of a constant, or a compile-time constant expression.

Syntax Rules and Behavior

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 4.11.

You can use symbolic names 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 data type of a parameter constant cannot be specified in a type declaration statement. Nor does the initial letter of the constant's name implicitly affect its data type.

Examples

```
PARAMETER  PI=3.1415927, DPI=3.14159265358979323 8D0
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 allow compatibility with PDP–11 FORTRAN.

An octal integer constant takes the following form:

```
"nn
```

nn
Is a string of digits in the range 0 to 7.

Examples

The following examples demonstrate valid and invalid octal integer constants and explain why the invalid ones are not valid:

Valid

"107

"177777

Invalid	Explanation
"108	Contains a digit outside the allowed range
"1377.	Contains a decimal point
"17777"	Contains a trailing quotation mark

These octal forms are not the same as the typeless octal constants discussed in Section 2.2.1.4. Integer constants in octal form have integer data type and are treated as integers.

A.6 /NOF77 Interpretation of the EXTERNAL Statement

By using the /NOF77 qualifier on the FORTRAN command line, you can obtain another interpretation of the EXTERNAL statement. This additional interpretation facilitates compatibility with older versions of FORTRAN due to the ANSI FORTRAN–77 interpretation being incompatible with the previous Standard and previous DIGITAL implementations.

The /NOF77 interpretation of the EXTERNAL statement combines the function of the INTRINSIC statement with that of the EXTERNAL statement discussed in Section 4.7. It is available only if the /NOF77 qualifier is specified on the FORTRAN command line.

The /NOF77 EXTERNAL statement lets you 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 takes the following form:

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

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 6.3 for information on FORTRAN library functions (intrinsic functions).

Syntax Rules and Behavior

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.

However, used as an argument, a complete function reference 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).

Example

The following example uses the /NOF77 EXTERNAL statement:

Main Program	Subprograms
`EXTERNAL SIN, COS, *TAN, SINDEG`	`SUBROUTINE TRIG(X,F,Y)`
.	`Y = F(X)`
.	`RETURN`
`CALL TRIG(ANGLE,SIN,SINE)`	`END`
.	
.	
`CALL TRIG(ANGLE,COS,COSINE)`	
.	
.	`FUNCTION TAN(X)`
`CALL TRIG(ANGLE,TAN,TANGNT)`	`TAN = SIN(X)/COS(X)`
.	`RETURN`
.	`END`
`CALL TRIG(ANGLED,SINDEG,SINE)`	
.	`FUNCTION SINDEG(X)`
.	`SINDEG = SIN(X*3.1459/180)`
.	`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

This appendix describes the following three character sets:

- FORTRAN
- ASCII
- Radix–50

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
Δor <TAB>	Spaceor 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 "16s" position. For example, the value of the character representing the equal sign is 3D.

Table B–1: ASCII Character Set

Row	Column 0	1	2	3	4	5	6	7
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 Control 2
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

ZK-7458-HC

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 corresponding values are given in Table B–2.

Table B–2: RADIX-50 Character Set with Comparative Values

Character	ASCII Octal Equivalent	Radix-50 Octal Value
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 following formula:

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

The values i, j, and k represent the code values of the three Radix–50 characters. Thus, the maximum Radix–50 value is as follows:

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

A Radix–50 constant takes the following form:

```
nRc1c2...cn
```

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.

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 (see Table 2–1). 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.

Examples

The following examples illustrate valid and invalid Radix–50 constants and explain why the invalid ones are not valid:

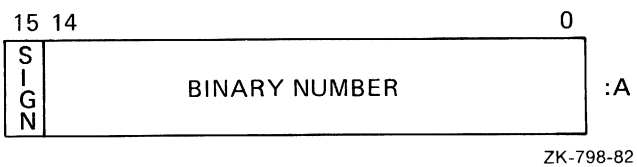
Valid	
4RABCD	
6RΔTOΔΔΔ	
Invalid	**Explanation**
4RDK0:	colon is not a Radix–50 character

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, which is the starting address of the represented data element.

C.1 INTEGER*2 Representation



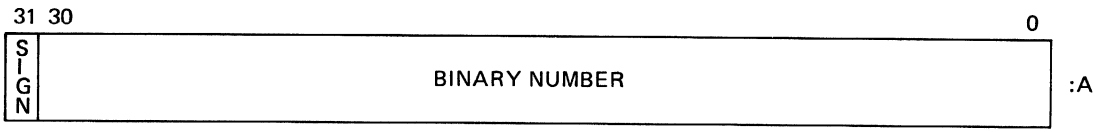
ZK-798-82

```
SIGN = 0(+), 1(-)
```

Integers are stored in a twos 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.2 INTEGER*4 Representation

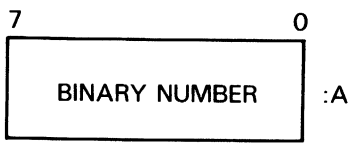


ZK-799-82

```
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 (–32768 to 32767), then the first word can be referenced as an INTEGER*2 value.

C.3 LOGICAL*1 (BYTE) Representation



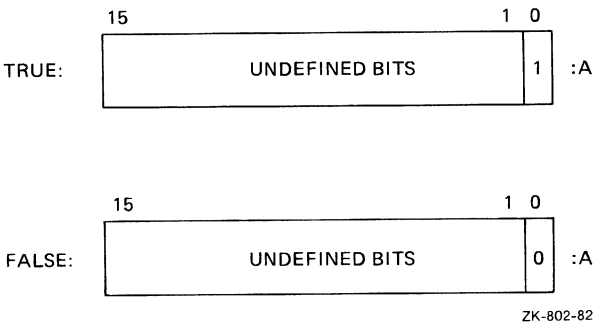
ZK-797-82

LOGICAL*1 (or BYTE) values are in the range –128 to 127.

C.4 LOGICAL*2 and LOGICAL*4 Representation

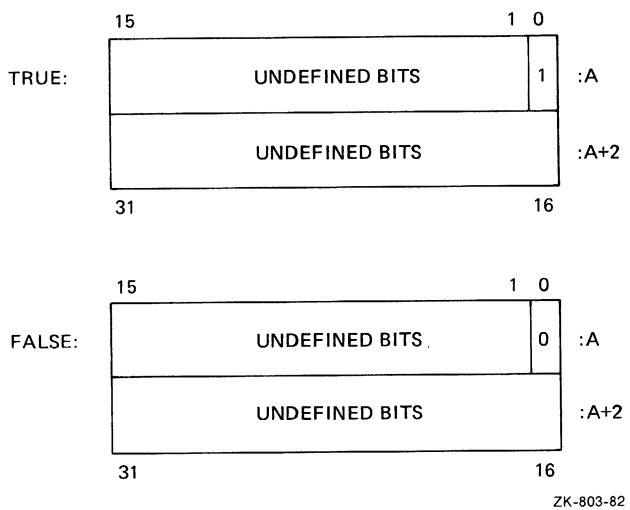
Logical values are stored in two (LOGICAL*2) or four (LOGICAL*4) contiguous bytes, starting on an arbitrary byte boundary. The low-order bit (bit 0) determines the value. If bit 0 is set, the value is .TRUE. If bit 0 is clear, the value is .FALSE.

LOGICAL*2



ZK-802-82

LOGICAL*4



ZK-803-82

C.5 Floating-Point Representations

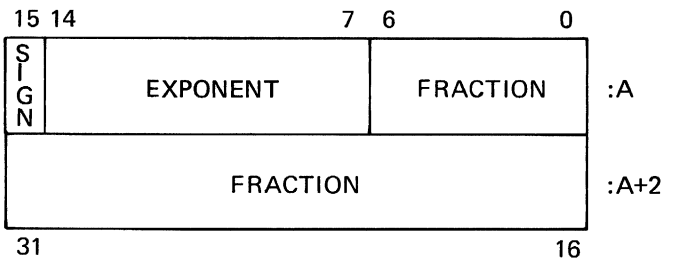
The exponent for the REAL*4 and REAL*8 (D_floating) formats is stored in binary excess 128 notation. Binary exponents from –127 to 127 are represented by the binary equivalents of 1 through 255.

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, or it is a reserved operand. (Refer to the *VAX FORTRAN User Manual* for an explanation of the representation of 0.0 and reserved operand faults.)

C.5.1 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.



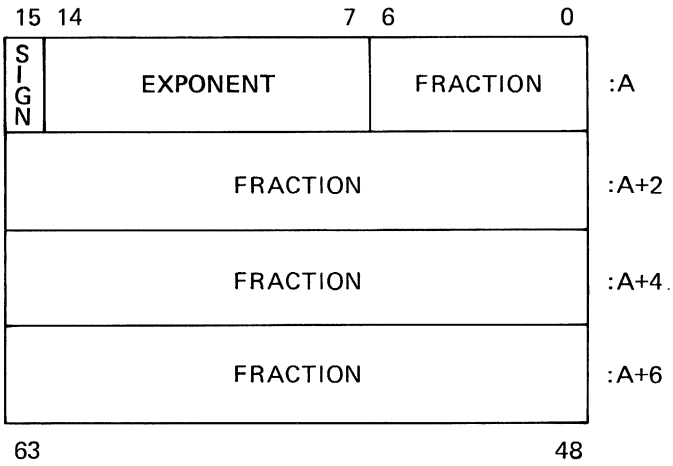
ZK-800-82

```
SIGN = 0(+), 1(-)
```

The form of REAL*4 (F_floating) data is sign magnitude, with bit 15 the sign bit, bits 14:7 an excess 128 binary exponent, and bits 6:0 and 31:16 a normalized 24-bit fraction with the redundant most significant fraction bit not represented. The value of F_floating data is in the approximate range: 0.29*10**–38 through 1.7*10**38. The precision is approximately one part in 2**23; that is, typically, seven decimal digits.

C.5.2 REAL*8 (D_floating)

REAL*8 (D_floating) data is eight contiguous bytes starting on an arbitrary byte boundary. Bits are labeled from the right, 0 through 63.



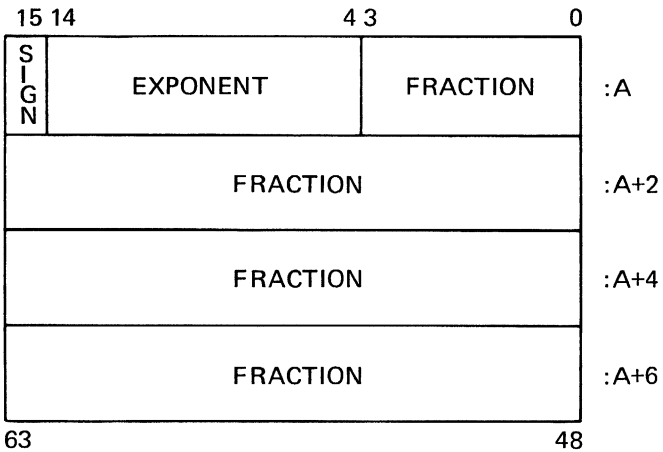
ZK-801-82

```
SIGN = 0(+), 1(-)
```

The form of REAL*8 (D_floating) data is identical to an F_floating real number, except for an additional 32 low-significance fraction bits. The exponent conventions and approximate range of values are the same as those for F_floating. The precision is approximately one part in 2**55; that is, typically, 16 decimal digits.

C.5.3 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.



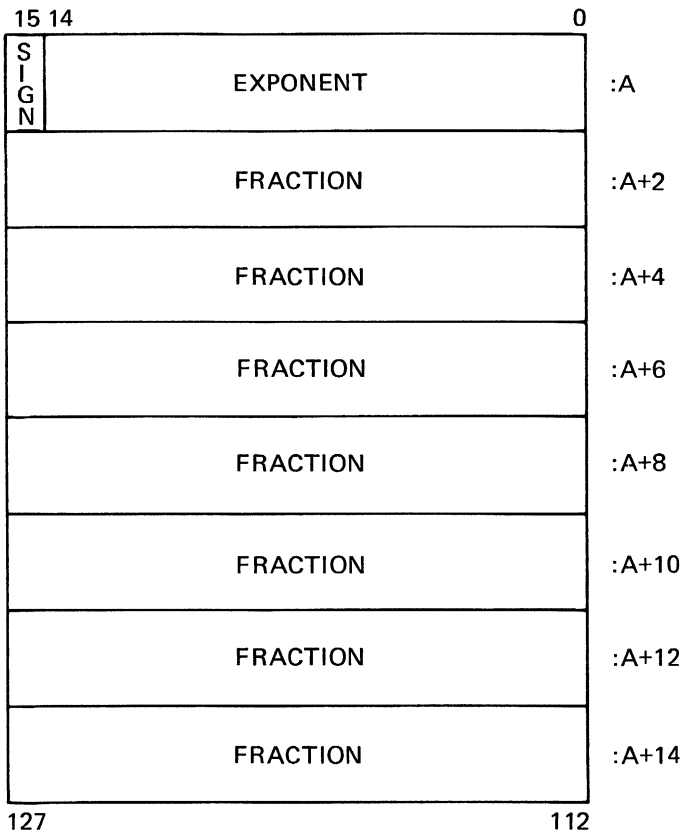
ZK-804-82

```
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.5.4 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.

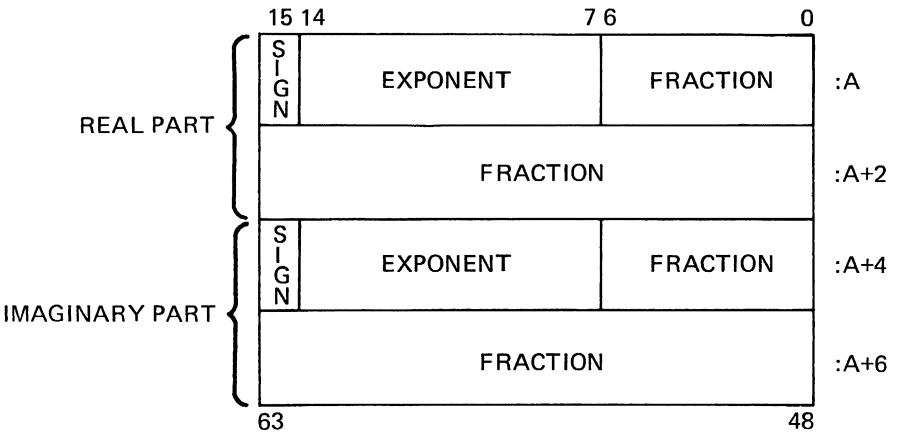



```
SIGN = 0(+), 1(-)
```

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*10**-4932 through 0.59*10**4932. The precision of H_floating data is approximately one part in 2**112; that is, typically, 33 decimal digits.

C.5.5 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.

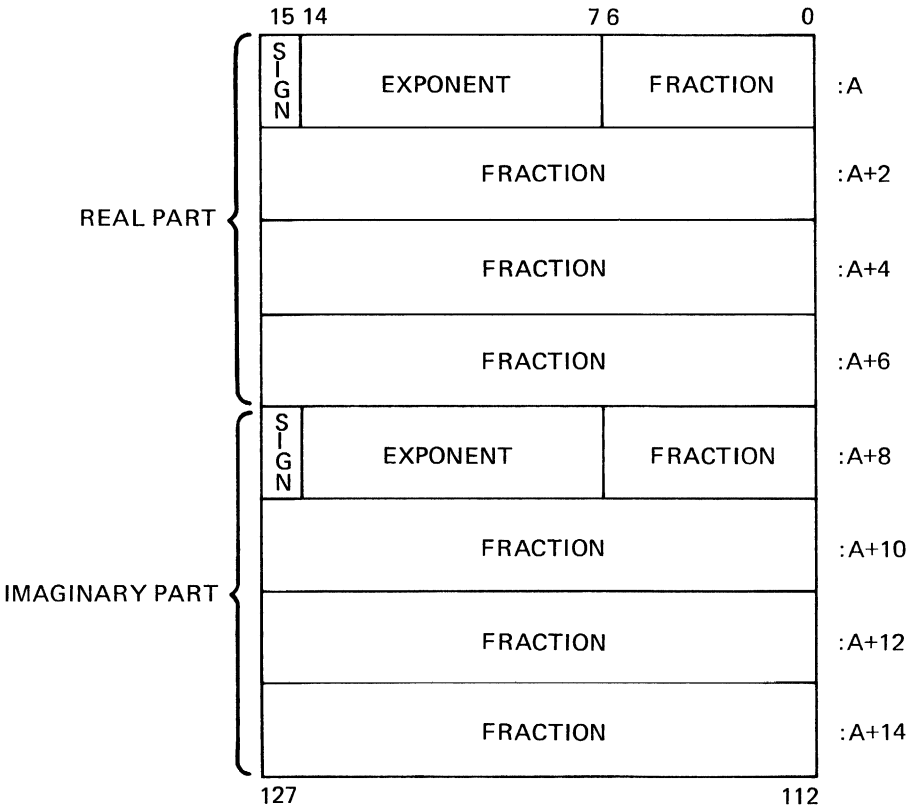


ZK-806-82

```
SIGN = 0(+), 1(-)
```

C.5.6 COMPLEX*16 (D_floating)

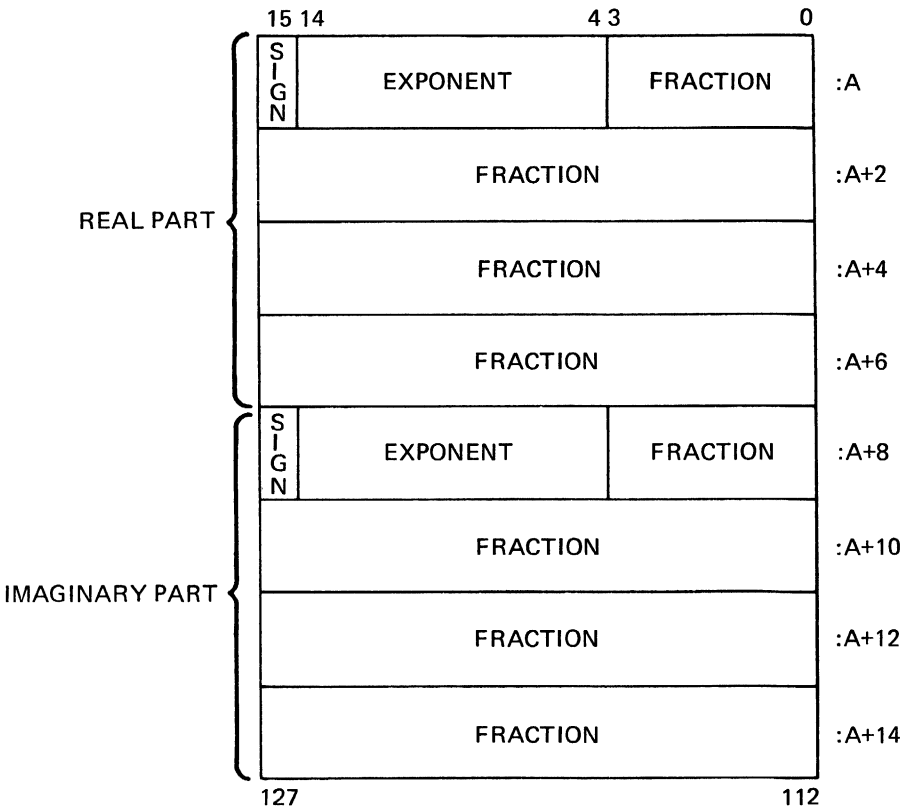
COMPLEX*16 (D_floating) data is 16 contiguous bytes aligned on an arbitrary byte boundary. The low-order eight bytes contain REAL*8 (D_floating) data that represents the real part of the complex data. The high-order eight bytes contain REAL*8 (D_floating) data that represents the imaginary part of the complex data.


15 14
7 6
0
REAL PART
IMAGINARY PART
SIGN
EXPONENT
FRACTION
:A
:A+2
:A+4
:A+6
:A+8
:A+10
:A+12
:A+14
127
112
ZK-807-82

SIGN = 0(+), 1(-)

C.5.7 COMPLEX*16 (G_floating)

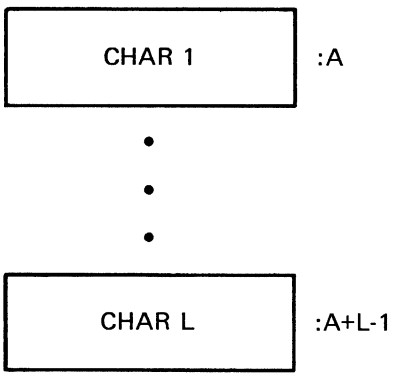
COMPLEX*16 (G_floating) data is 16 contiguous bytes aligned on an arbitrary byte boundary. The low-order eight bytes contain REAL*8 (G_floating) data that represents the real part of the complex data. The high-order eight bytes contain REAL*8 (G_floating) data that represents the imaginary part of the complex data.



ZK-808-82

C.6 Character Representation

A character string is a contiguous sequence of bytes in memory.



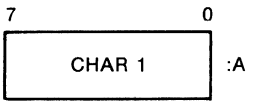
ZK-809-82

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.

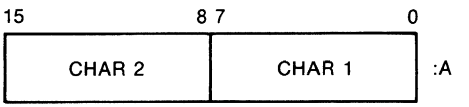
C.7 Hollerith Representation

Hollerith constants are stored internally, one character per byte.

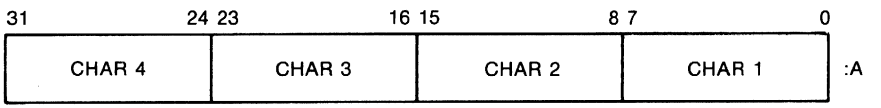
1 Byte



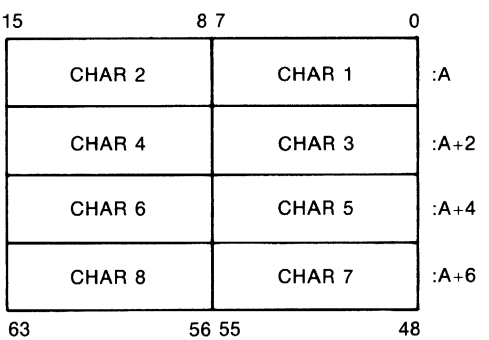
2 Bytes



4 Bytes



8 Bytes



ZK-810-82

Appendix D

VAX FORTRAN Language Summary

This appendix summarizes VAX FORTRAN expression operators (Table D–1), statements (Table D–2), and intrinsic functions (Table D–3). It also describes the system subroutines provided with VAX FORTRAN (Section D.4) and discusses the intrinsic functions available for manipulating bits in integer data items (Section D.5).

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. .GE.	Greater than Greater than or equal to	Arithmetic, logical, or character expressions (all relational operators have equal precedence)
	.LT.	Less than	

Table D–1 (Cont.): Expression Operators

Data Type	Operator	Operation	Operates Upon
	.LE.	Less than or equal to	
	.EQ.	Equal to	
	.NE.	Not equal to	
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.

Table D–2: VAX FORTRAN Language

Statement Summary

ACCEPT

See READ statements (sequential access).

See Section 7.5.

ASSIGN s TO v

s is the label of an executable statement or a FORMAT statement.

v is an integer variable name.

Associates the statement label s with the integer variable v for later use as a format specifier or in an assigned GO TO statement.

See Section 3.4.

Assignment Statement

v = e

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.

See Sections 3.1 – 3.3.

BACKSPACE ([UNIT=]u[,ERR=s][,IOSTAT=ios])
BACKSPACE u

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 u by one record.

See Section 9.5.

BLOCK DATA [nam]

nam is a symbolic name.

The BLOCK DATA statement specifies the subprogram that follows as a BLOCK DATA subprogram.

See Section 4.1.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

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

sub — 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 *s or &s, where s is the label of an executable statement.

The CALL statement calls the subroutine subprogram with the name specified by sub, passing the actual arguments (a) to replace the dummy arguments in the subroutine definition.

See Sections 5.1 and 6.2.3.

CDEC$ IDENT string

string — is a group of up to 31 printable characters delimited by apostrophes.

The IDENT directive specifies a string that identifies an object module.

See Section 10.3.1.

CDEC$ PSECT /common-name/ attr [,attr]...

common-name — is the name of the common block, which must be preceded and followed by a slash.

attr — is one of the following attributes: LCL (local scope); GBL (global scope); [NO]WRT (writability or no writability); [NO]SHR (shareability or no shareability); or ALIGN = val (alignment for the common block, val must be a constant ranging from 0 thru 9).

The PSECT directive modifies certain attributes of a common block.

See Section 10.3.2.

CDEC$ SUBTITLE string

string — is a group of up to 31 printable characters delimited by apostrophes.

The SUBTITLE directive creates a header in a listing's subtitle field.

See Section 10.3.3.

CDEC$ TITLE string

string — is a group of up to 31 printable characters delimited by apostrophes.

The TITLE directive creates a header in a listing's title field.

See Section 10.3.3.

CLOSE ([UNIT=]u[,p][,ERR=s][,IOSTAT=ios])

u — is a logical unit specifier.

p is one of the following parameters:

$$\left\{\begin{array}{l}\text{STATUS}\\\text{DISPOSE}\\\text{DISP}\end{array}\right\} = \left\{\begin{array}{l}\text{'SAVE'}\\\text{'KEEP'}\\\text{'DELETE'}\\\text{'PRINT'}\\\text{'SUBMIT'}\\\text{'PRINT/DELETE'}\\\text{'SUBMIT/DELETE'}\end{array}\right\}$$

s is the label of an executable statement.

ios is an I/O status specifier.

The CLOSE statement closes the specified file.

See Section 9.2.

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

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.

See Section 4.2.

CONTINUE

The CONTINUE statement transfers control to the next executable statement.

See Section 5.2.

CPAR$ CONTEXT_SHARED syname [,syname]...

syname is the name of a variable, array, or record declared within the routine.

The CPAR$ CONTEXT_SHARED directive specifies shared memory locations for symbols declared in the routines compiled with the /PARALLEL qualifier.

See Section 10.2.1.

CPAR$ CONTEXT_SHARED_ALL

The CPAR$ CONTEXT_SHARED_ALL directive reinforces the context-shared default of symbols in routines compiled with the /PARALLEL qualifier.

See Section 10.2.2.

CPAR$ DO_PARALLEL [count]

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

count — is a numeric expression specifying the number of iterations in each set distributed to processors running the parallel DO-loop. The number must be able to be evaluated as a positive, non-zero integer.

The DO_PARALLEL directive enables parallel processing of the DO-loop that follows it.

See Section 10.2.3.

CPAR$ LOCKON lck-var
(critical region)
CPAR$ LOCKOFF lck-var

lck-var — is a scalar memory reference with a LOGICAL*4 data type.

The LOCKON and LOCKOFF directives enclose a critical region of executable code within a parallel DO-loop and permit only one process at a time to execute the region.

See Section 10.2.4.

CPAR$ PRIVATE name [,name]...

name — is the name of a symbol or a common block (preceded and followed by a slash).

The CPAR$ PRIVATE directive specifies the common blocks or symbols that must be private for each process that runs a parallel DO-loop.

See Section 10.2.5.

CPAR$ PRIVATE_ALL

The CPAR$ PRIVATE_ALL directive forces all common blocks and symbols to have PRIVATE defaults in routines compiled with the /PARALLEL qualifier.

See Section 10.2.6.

CPAR$ SHARED /[cb]/[,/[cb]/]...

cb — is the name of a common block.

The CPAR$ SHARED directive specifies the common blocks to be shared by each process that runs a parallel DO-loop.

See Section 10.2.7.

CPAR$ SHARED_ALL

The CPAR$ SHARED_ALL directive forces all common blocks to have SHARED defaults in routines compiled with the /PARALLEL qualifier.

See Section 10.2.8.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

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

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 constants.

clist — is a list of one or more constants separated by commas, each optionally preceded by j*, where j is a nonzero, unsigned integer constant.

The DATA statement initially stores elements of clist in the corresponding elements of nlist.

See Section 4.3.

Data Type Declaration

See Type Declaration.

DECODE (c,f,b[,ERR=s][,IOSTAT=ios]) [list]

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 list, converted according to format specification f.

See Section A.1.

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

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.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

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.

See Section A.2.

DELETE([UNIT=]u[,REC=r][,ERR=s][,IOSTAT=ios])
DELETE (u'r[,ERR=s][,IOSTAT=ios])

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.

The DELETE statement deletes records from relative or indexed files.

See Section 9.7.

DICTIONARY 'cdd-path[/[NO]LIST] '

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.

The DICTIONARY statement extracts a data definition from the VAX/VMS Common Data Dictionary, translates it to FORTRAN source code, and includes it in a FORTRAN source program.

See Section 1.4.1.

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

a(d) is an array declarator.

a is an array name.

d is the lower (optional) and upper bounds of the array. It takes the form [dl:]du where dl is the lower bound and du is the upper bound.

The DIMENSION statement specifies storage space requirements for arrays.

See Section 4.5.

DO [s[,]] v=e1,e2[,e3]

s is the label of an executable statement. VAX FORTRAN allows the statement label to be omitted.

v is a variable name.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

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.

The DO statement executes the DO loop by performing the following steps:

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:
   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.

See Section 5.3.

DO [s[,]] WHILE (e)

s	is the label of an executable statement. VAX FORTRAN allows the statement label to be omitted.
e	is a logical expression.

The DO WHILE statement is similar to the DO statement, but DO WHILE executes as long as the logical expression contained in the statement continues to be true, instead of for a specified number of iterations.

See Section 5.3.2.

ELSE

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.

See Section 5.7.3.

ELSE IF (e) THEN

e	is a logical expression.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

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.

See Section 5.7.3.

ENCODE (c,f,b[,ERR=s][,IOSTAT=ios]) [list]

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.

See Section A.1.

END

The END statement marks the end of a program unit.

See Section 5.5.

END DO

The END DO statement marks the end of the body of a DO loop, and may be used in place of a labeled statement.

See Section 5.4.

END IF

The END IF statement marks the end of a block IF construct.

See Section 5.7.3.

END MAP

The END MAP statement marks the end of a field declaration or a series of field declarations in a UNION.

See Section 4.15.3.

END STRUCTURE

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

The END STRUCTURE statement marks the end of a structure declaration.

See Section 4.15.1.

END UNION

The END UNION statement marks the end of a union declaration.

See Section 4.15.3.

ENDFILE ([UNIT=]u[,ERR=s][,IOSTAT=ios])
ENDFILE u

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 u.

See Section 9.6.

ENTRY nam[([p[,p]...])]

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.

See Section 6.2.4.

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

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.

See Section 4.6.

EXTERNAL v[,v]...
EXTERNAL *v[*v]...

v is a subprogram name.

* is used only if /NOF77 is specified.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

The EXTERNAL statement defines the names specified as user-defined subprograms.

See Sections 4.7 and A.6.

FIND ([UNIT=]u,REC=r[,ERR=s][,IOSTAT=ios])
FIND (u'r[,ERR=s][,IOSTAT=ios])

u	is a logical unit specifier.
r	is a direct access record number.
s	is a label of an executable statement.
ios	is an I/O status specifier.

The FIND statement positions the file on logical unit u to record r and sets the associated variable to record number r.

See Section A.3.

FORMAT (format-spec[,...])

format-spec	is a list of field descriptors and field separators.

The FORMAT statement describes the format in which one or more records are to be transmitted; a statement label must be specified on the FORMAT statement.

See Chapter 8.

[typ] FUNCTION nam[*m][([p[,p]...])]

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.

See Section 6.2.2.

GO TO s

s	is a label of an executable statement.

This GO TO statement transfers control to statement number s.

See Section 5.6.1.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

GO TO (slist)[,] e

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.

See Section 5.6.2.

GO TO v[[,](slist)]

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.

See Section 5.6.3.

IF (e) s1,s2,s3

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).

See Section 5.7.1.

IF (e) st

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.

See Section 5.7.2.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

IF e1 **THEN**
    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.

See Section 5.7.3.

IMPLICIT typ(a[,a]...)[,typ(a[,a]...)]...
IMPLICIT NONE

typ is a data type specifier.

a is either a single letter, or two letters in alphabetical order, separated by a hyphen (for example, X-Y).

NONE inhibits the implicit type declaration of variables in the module.

The IMPLICIT statement implicitly declares the data types of variables within program units. The element a represents a single letter or a range of letters whose presence as the initial letter of a variable specifies the variable to be of that data type.

IMPLICIT NONE and IMPLICIT must not be used in the same program unit.

See Section 4.8.

INCLUDE 'file-spec[/[NO]LIST] '
INCLUDE '[file-spec](module-name)[/[NO]LIST] '

file-spec is a character constant that specifies the file to be included.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

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.

The INCLUDE statement includes the source statements in the compilation from the file or module specified.

See Section 1.4.2.

INQUIRE (par[,par]...)

par — is a keyword specification having the form:

```
key = value
```

key — is a keyword (see the list of keywords and values that follows).

value — depends on the keyword.

Keyword	Values
Inputs	
FILE	fin
UNIT	e
DEFAULTFILE	fin
Outputs	
ACCESS	cv
BLANK	cv
CARRIAGECONTROL	cv
DIRECT	cv
ERR	s

Table D–2 (Cont.): VAX FORTRAN Language Statement Summary

Keyword	Values
Outputs (*continued*)	
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.

See Section 9.3.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

INTRINSIC v[,v]...

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.

See Section 4.9.

Map Declaration

See Union Declaration.

NAMELIST /group-name/ namelist[[,] /group-name/ namelist]...

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.

See Section 4.10.

OPEN (par[,par]...)

par — is a keyword specification in one of the following forms:

```
keyword
keyword = value
```

keyword — is a keyword (see the list of keywords and values that follows).

value — depends on the keyword.

Table D–2 (Cont.): VAX FORTRAN Language Statement Summary

Keyword	Values
ACCESS	'SEQUENTIAL' 'DIRECT' 'KEYED' 'APPEND'
ASSOCIATEVARIABLE	v
BLANK	'NULL' 'ZERO'
BLOCKSIZE	e
BUFFERCOUNT	e
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'
DEFAULTFILE	c
DISP	(same as DISPOSE)
DISPOSE	'KEEP' or 'SAVE' 'DELETE' 'PRINT' 'SUBMIT' 'SUBMIT/DELETE' 'PRINT/DELETE'
ERR	s
EXTENDSIZE	e
FILE	c

Table D–2 (Cont.): VAX FORTRAN Language Statement Summary

Keyword	Values
FORM	'FORMATTED' 'UNFORMATTED'
INITIALSIZE	e
IOSTAT	v
KEY	keyspec
MAXREC	e
NAME	(same as FILE)
NOSPANBLOCKS	—
ORGANIZATION	'SEQUENTIAL' 'RELATIVE' 'INDEXED'
READONLY	—
RECL	e
RECORDSIZE	(same as RECL)
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED' 'STREAM' 'STREAM_CR' 'STREAM_LF'
SHARED	—
STATUS	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'
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[:dt[:dr]].

e1	is first byte position of the key.
e2	is last byte position of the key.
dt	is the data type, INTEGER or CHARACTER.
dr	is the direction of the key, ASCENDING or DESCENDING.

The OPEN statement opens a file on the specified logical unit according to the parameters specified by the keywords.

See Section 9.1.

OPTIONS qualifier[qualifier...]

qualifier is one of the following:

```
/NOCHECK
/CHECK= { ALL                                                  }
        { ([NO]OVERFLOW, [NO]BOUNDS, [NO]UNDERFLOW)            }
        { NONE                                                 }
/[NO]EXTEND_SOURCE
/[NO]F77
/[NO]G_FLOATING
/[NO]I4
```

The OPTIONS statement overrides the command line qualifiers for a single program unit.

See Section 1.4.3.

PARAMETER (p=c[,p=c]...)

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.

See Sections 4.11 and A.4.

PAUSE [disp]

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.

See Section 5.8.

PRINT

See WRITE statements (sequential access).

See Section 7.6.

PROGRAM nam

nam — is a program name.

The PROGRAM statement specifies a name for the main program.

See Section 4.12.

Read Statements—Formatted Sequential Access

READ ([UNIT=]u,[FMT=]f[,ERR=s][,IOSTAT=ios][,END=s]) [list]
READ f[,list]
ACCEPT f[,list]

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.

These input statements read one or more logical records from unit u and assign values to the elements in the list. The records are converted according to the format specifier (f).

See Section 7.2.1.1 and 7.5.

Read Statements—List-Directed Sequential Access

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

READ ([UNIT=]u,[FMT=]*[,ERR=s][,IOSTAT=ios][,END=s]) [list]
READ *[,list]
ACCEPT *[,list]

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.

These input statements read one or more logical records from unit u and assign values to the elements in the list. The records are converted according to the data type of the list element.

See Section 7.2.1.2 and 7.5.

Read Statements—Namelist-Directed Sequential Access

READ ([UNIT=]u,[NML=]nl[,ERR=s][,IOSTAT=ios][,END=s])
READ nl
ACCEPT nl

u	is a logical unit specifier.
nl	is a namelist group-name.
s	is a label of an executable statement.
ios	is an I/O status specifier.

These input statements read one or more logical records from unit u and assign values to specified namelist entities. The records are converted according to the data type of the namelist entities.

See Sections 7.2.1.3 and 7.5.

Read Statements—Unformatted Sequential Access

READ ([UNIT=]u[,ERR=s][,IOSTAT=ios][,END=s]) [list]

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.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

See Section 7.2.1.4.

Read Statements—Formatted Direct Access

READ ([UNIT=]u,[FMT=]f,REC=r[,ERR=s][,IOSTAT=ios]) [list]
READ (u'r,[FMT=]f[,ERR=s][,IOSTAT=ios]) [list]

u	is a logical unit specifier.
r	is a record 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 READ statement reads record r from unit u and assigns values to the elements in the list. The record is converted according to f.

See Section 7.2.2.1.

Read Statements—Unformatted Direct Access

READ ([UNIT=]u,REC=r[,ERR=s][,IOSTAT=ios]) [list]
READ (u'r[,ERR=s][,IOSTAT=ios]) [list]

u	is a logical unit specifier.
r	is a record 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.

See Section 7.2.2.2.

Read Statements—Formatted Indexed and Unformatted Indexed

Formatted indexed READ statement:

READ ([UNIT=]u,[FMT=]f,keyspec[,KEYID=kn][,ERR=s] [,IOSTAT=ios]) [list]

Unformatted indexed READ statement:

READ ([UNIT=]u,keyspec[,KEYID=kn][,ERR=s] [,IOSTAT=ios]) [list]

u	is a logical unit specifier.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

f	is a format specifier.
keyspec	is a key specifier (see Section 7.1.1.6).
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.

These input statements read one or more logical records specified by key value, and assign values to the elements in the list.

See Sections 7.2.3.1 and 7.2.3.2.

Read Statements—Formatted Internal and List-Directed Internal

Formatted internal READ statement:

READ (intu, fmt [,iostat][,err][,end]) [iolist]

List-directed internal READ statement:

READ (intu,*[,iostat][,err][,end]) [iolist]

intu	is an internal file specifier.
fmt	is a format specifier.
*	is a list-directed formatting specifier.
iostat	is an I/O status specifier.
err, end	are transfer-of control specifiers.
iolist	is the I/O list specifier.

These input statements read into elements in the list. They read one or more internal records containing character strings, converting in accordance with the format specification.

See Section 7.2.4.

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 or array names, or both; separated by commas.

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.

See Section 4.13.

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.

See Section 5.9.

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.

See Section 9.4.

REWRITE Statement—Formatted Indexed and Unformatted Indexed

Formatted indexed REWRITE statement:

REWRITE ([UNIT=]u,[FMT=]f[,ERR=s][,IOSTAT=ios]) [list]

Unformatted indexed REWRITE statement:

REWRITE ([UNIT=]u[,ERR=s][,IOSTAT=ios]) [list]

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.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

See Section 7.4.

SAVE [a[,a]...]

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.

See Section 4.14.

Statement Function

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 p as dummy arguments. When referred to, the expression is evaluated using the actual arguments in the function call.

See Section 6.2.1.

STOP [disp]

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.

See Section 5.10.

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.

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 (records) are created when you use a RECORD statement containing the name of a previously declared structure.

See Section 4.15.1.

SUBROUTINE nam[([p[,p]...])]

nam — is a subroutine name.

p — is a dummy argument or an alternate return specifier (*).

The SUBROUTINE statement begins a subroutine subprogram, indicating the program name and any dummy argument names (p).

See Section 6.2.3.

TYPE

See WRITE statements (sequential access).

See Section 7.6.

Type Declarations—Character and Numeric

Type Declaration (Character):

CHARACTER[*len[,]] v[*len][/clist/][,v[*len][/clist/]...

len — specifies the length of the character data elements; or it can be an asterisk enclosed in parentheses.

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

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).

See Section 4.4.2.

Type Declaration (Numeric):

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

type [*n]v[*n][clist][,v[*n][/clist/]]

type	is one of the following data type specifiers: BYTE, LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or DOUBLE COMPLEX.
*n	is an integer that specifies the byte length of v. It must be an acceptable length for the entity (see Table 2–1); *n is not valid for DOUBLE PRECISION and DOUBLE COMPLEX data types.
v	is a variable name, array name, function or function entry name, or an array declarator.
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).

See Section 4.4.1.

Union Declaration

```
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.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

See Section 4.15.3.

UNLOCK ([UNIT=]u[,ERR=s][,IOSTAT=ios])
UNLOCK u

u — is a logical unit specifier.

s — is a label of an executable statement.

ios — is an I/O status specifier.

The UNLOCK statement frees a previously locked record in the file connected to logical unit u.

See Section 9.8.

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

a(d) — is an array declarator.

a — is an array name.

d — is the lower (optional) and upper bounds of the array. It takes the form [dl:]du where dl is the lower bound and du is the upper bound.

The VIRTUAL statement has the same effect as the DIMENSION statement and is included for compatibility with PDP–11 FORTRAN.

See Section 4.5.

VOLATILE nlist

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.

See Section 4.16.

Write Statements—Formatted Sequential Access

WRITE ([UNIT=]u,[FMT=]f[,ERR=s][,IOSTAT=ios]) [list]
PRINT f[,list]
TYPE f[,list]

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.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

These output statements write one or more logical records to unit u containing the values of the elements in the list. The records are converted according to f.

For information about the formatted sequential access WRITE statement, see Section 7.3.1.1. For information about formatted sequential access PRINT and TYPE statements, see Section 7.6.

Write Statements—List-Directed Sequential Access

WRITE ([UNIT=]u,[FMT=]*[,ERR=s][,IOSTAT=ios]) [list]
PRINT *[,list]
TYPE *[,list]

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.

These output statements write 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.

For information about the list-directed sequential access WRITE statement, see Section 7.3.1.2. For information about list-directed sequential access PRINT and TYPE statements, see Section 7.6.

Write Statements—Namelist-Directed Sequential Access

WRITE ([UNIT=]u,[NML=]nl[,ERR=s] [,IOSTAT=ios])
PRINT nl
TYPE nl

u	is a logical unit specifier.
nl	is a namelist group-name.
s	is a label of an executable statement.
ios	is an I/O status specifier.

These output statements write 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.

For information about the namelist-directed sequential access WRITE statement, see Section 7.3.1.3. For information about namelist-directed sequential access PRINT and TYPE statements, see Section 7.6.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary

Write Statements—Unformatted Sequential Access

WRITE ([UNIT=]u[,ERR=s][,IOSTAT=ios]) [list]

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 u containing the values of the elements in the list.

See Section 7.3.1.4.

Write Statements—Formatted Direct Access

WRITE ([UNIT=]u,[FMT=]f,REC=r[,ERR=s][,IOSTAT=ios]) [list]
WRITE (u'r,f[,ERR=s][,IOSTAT=ios]) [list]

u	is a logical unit specifier.
r	is a record 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.

These WRITE statements write the values of the elements of the list to record r on unit u. The record is converted according to f.

See Section 7.3.2.1

Write Statements—Unformatted Direct Access

WRITE ([UNIT=]u,REC=r[,ERR=s][,IOSTAT=ios]) [list]
WRITE (u'r[,ERR=s][,IOSTAT=ios]) [list]

u	is a logical unit specifier.
r	is a record specifier.
s	is a label of an executable statement.
ios	is an I/O status specifier.
list	is an I/O list.

These WRITE statements write record r to unit u containing the values of the elements in the list.

Table D–2 (Cont.): VAX FORTRAN Language

Statement Summary
See Section 7.3.2.2.
Write Statements—Formatted Internal and List-Directed Internal
Formatted internal WRITE statement:
WRITE (intu,[FMT=]f[,ERR=s][,IOSTAT=ios]) [list]
List-directed internal WRITE statement:
WRITE (intu,[FMT=]*[,ERR=s] [,IOSTAT=ios]) [list]
intu 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.
These WRITE statements write elements in the list to the internal file specified by intu. The formatted internal WRITE statement converts the elements to character strings in accordance with the format specification.
See Section 7.3.4.

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 6.3 for more information about intrinsic functions. For descriptions of the intrinsic function algorithms, refer to the *VMS RTL Mathematics (MTH$) Manual.*

Table D–3: VAX FORTRAN Intrinsic Functions

Functions	No. 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

[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 2*pi. The argument of SIND, COSD, or TAND must be in degrees. The argument is treated modulo 360.

Table D–3 (Cont.): VAX FORTRAN Intrinsic Functions

Functions	No. of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Cosine[3]	1	COS	COS	REAL*4	REAL*4
Cos a			DCOS	REAL*8	REAL*8
			QCOS	REAL*16	REAL*16
			CCOS	COMPLEX*8	COMPLEX*8
			CDCOS	COMPLEX*16	COMPLEX*16
Cosine[3](degree)	1	COSD	COSD	REAL*4	REAL*4
Cos a			DCOSD	REAL*8	REAL*8
			QCOSD	REAL*16	REAL*16
Tangent[3]	1	TAN	TAN	REAL*4	REAL*4
Tan a			DTAN	REAL*8	REAL*8
			QTAN	REAL*16	REAL*16
Tangent[3](degree)	1	TAND	TAND	REAL*4	REAL*4
Tan a			DTAND	REAL*8	REAL*8
			QTAND	REAL*16	REAL*16
Arc Sine[4,5]	1	ASIN	ASIN	REAL*4	REAL*4
Arc Sin a			DASIN	REAL*8	REAL*8
			QASIN	REAL*16	REAL*16
Arc Sine (degree)	1	ASIND	ASIND	REAL*4	REAL*4
Arc Sin a			DASIND	REAL*8	REAL*8
			QASIND	REAL*16	REAL*16
Arc Cosine[4,5]	1	ACOS	ACOS	REAL*4	REAL*4
Arc Cos a			DACOS	REAL*8	REAL*8
			QACOS	REAL*16	REAL*16
Arc Cosine (degree)	1	ACOSD	ACOSD	REAL*4	REAL*4
Arc Cos a			DACOSD	REAL*8	REAL*8
			QACOSD	REAL*16	REAL*16

[3]The argument of SIN, DSIN, QSIN, COS, DCOS, QCOS, TAN, DTAN, or QTAN must be in radians. The argument is treated modulo 2*pi. 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.

Table D–3 (Cont.): VAX FORTRAN Intrinsic Functions

Functions	No. of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Arc Tangent[5]	1	ATAN	ATAN	REAL*4	REAL*4
Arc Tan a			DATAN	REAL*8	REAL*8
			QATAN	REAL*16	REAL*16
Arc Tangent[5,7](degree)	1	ATAND	ATAND	REAL*4	REAL*4
Arc Tan a			DATAND	REAL*8	REAL*8
			QATAND	REAL*16	REAL*16
Arc Tangent[5,6]	2	ATAN2	ATAN2	REAL*4	REAL*4
Arc Tan a_1/a_2			DATAN2	REAL*8	REAL*8
			QATAN2	REAL*16	REAL*16
Arc Tangent[5,7](degree)	2	ATAN2D	ATAN2D	REAL*4	REAL*4
Arc Tan a_1/a_2			DATAN2D	REAL*8	REAL*8
			QATAN2D	REAL*16	REAL*16
Hyperbolic Sine	1	SINH	SINH	REAL*4	REAL*4
Sinh a			DSINH	REAL*8	REAL*8
			QSINH	REAL*16	REAL*16
Hyperbolic Cosine	1	COSH	COSH	REAL*4	REAL*4
Cosh a			DCOSH	REAL*8	REAL*8
			QCOSH	REAL*16	REAL*16
Hyperbolic Tangent	1	TANH	TANH	REAL*4	REAL*4
Tanh a			DTANH	REAL*8	REAL*8
			QTANH	REAL*16	REAL*16

[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 pi 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: $-pi < result < pi$.

[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 ATAN2D, DTAN2D, QATAN2D is: -180 degrees $<$ result $<$ 180 degrees.

Table D–3 (Cont.): VAX FORTRAN Intrinsic Functions

Functions	No. of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Absolute Value[8]	1	ABS	IIABS	INTEGER*2	INTEGER*2
|a|			JIABS	INTEGER*4	INTEGER*4
			ABS	REAL*4	REAL*4
			DABS	REAL*8	REAL*8
			QABS	REAL*16	REAL*16
			CABS	COMPLEX*8	REAL*4
			CDABS	COMPLEX*16	REAL*8
		IABS	IIABS	INTEGER*2	INTEGER*2
			JIABS	INTEGER*4	INTEGER*4
Truncation[9,10]	1	INT	IINT	REAL*4	INTEGER*2
[a]			JINT	REAL*4	INTEGER*4
			IIDINT	REAL*8	INTEGER*2
			JIDINT	REAL*8	INTEGER*4
			IIQINT	REAL*16	INTEGER*2
			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

[8]The absolute value of a complex number, (X,Y), is the real value SQRT (X**2 + Y**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]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.

Table D–3 (Cont.): VAX FORTRAN Intrinsic Functions

Functions	No. of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Nearest Integer[9,10] [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		
		ANINT	ANINT	REAL*4	REAL*4
			DNINT	REAL*8	REAL*8
			QNINT	REAL*16	REAL*16
Zero-Extend Functions	1	ZEXT	IZEXT	LOGICAL*1	INTEGER*2
				LOGICAL*2	
				LOGICAL*2	
			JZEXT	LOGICAL*1	INTEGER*4
				LOGICAL*2	
				LOGICAL*4	
				INTEGER*2	
				INTEGER*4	
Conversion to REAL*4[11]	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

[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]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.

[11]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.

Table D–3 (Cont.): VAX FORTRAN Intrinsic Functions

Functions	No. of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Conversion to REAL*8[11]	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
Conversion to REAL*16[11]	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[10,11] (REAL*4-to-integer conversion)	1	IFIX	IIFIX	REAL*4	INTEGER*2
			JIFIX	REAL*4	INTEGER*4
Float[11] (Integer-to-REAL*4 conversion)	1	FLOAT	FLOATI	INTEGER*2	REAL*4
			FLOATJ	INTEGER*4	REAL*4
REAL*8 Float[11] (Integer-to-REAL*8 conversion)	1	DFLOAT	DFLOTI	INTEGER*2	REAL*8
			DFLOTJ	INTEGER*4	REAL*8
REAL*16 Float (Integer-to-REAL*16 conversion)		QFLOAT	—	INTEGER*2	REAL*16
				INTEGER*4	REAL*16

[10]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.

[11]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.

Table D–3 (Cont.): VAX FORTRAN Intrinsic Functions

Functions	No. of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Conversion to COMPLEX*8, or COMPLEX*8 from Two Arguments[12]	1,2	CMPLX	—	INTEGER*2	COMPLEX*8
	1,2		—	INTEGER*4	COMPLEX*8
	1,2		—	REAL*4	COMPLEX*8
	1,2		—	REAL*8	COMPLEX*8
	1,2		—	REAL*16	COMPLEX*8
	1		—	COMPLEX*8	COMPLEX*8
	1		—	COMPLEX*16	COMPLEX*8
Conversion to COMPLEX*16, or COMPLEX*16 from Two Arguments[12]	1,2	DCMPLX	—	INTEGER*2	COMPLEX*16
	1,2		—	INTEGER*4	COMPLEX*16
	1,2		—	REAL*4	COMPLEX*16
	1,2		—	REAL*8	COMPLEX*16
	1,2		—	REAL*16	COMPLEX*16
	1		—	COMPLEX*8	COMPLEX*16
	1		—	COMPLEX*16	COMPLEX*16
Real Part of Complex	1	—	REAL	COMPLEX*8	REAL*4
			DREAL	COMPLEX16	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*4s a_1*a_2	2	—	DPROD	REAL*4	REAL*8

[12]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.)

Table D–3 (Cont.): VAX FORTRAN Intrinsic Functions

Functions	No. of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Maximum[10]	n	MAX	IMAX0	INTEGER*2	INTEGER*2
$\max(a_1,a_2, \ldots a_n)$			JMAX0	INTEGER*4	INTEGER*4
			AMAX1	REAL*4	REAL*4
(returns the value			DMAX1	REAL*8	REAL*8
from among the			QMAX1	REAL*16	REAL*16
argument list;		MAX0	IMAX0	INTEGER*2	INTEGER*2
there must be at			JMAX0	INTEGER*4	INTEGER*4
least two		MAX1	IMAX1	REAL*4	INTEGER*2
arguments)			JMAX1	REAL*4	INTEGER*4
		AMAX0	AIMAX0	INTEGER*2	REAL*4
			AJMAX0	INTEGER*4	REAL*4
Minimum [10]	n	MIN	IMIN0	INTEGER*2	INTEGER*2
$\min(a_1,a_2, \ldots a_n)$			JMIN0	INTEGER*4	INTEGER*4
			AMIN1	REAL*4	REAL*4
(returns the			DMIN1	REAL*8	REAL*8
minimum value			QMIN1	REAL*16	REAL*16
from among		MIN0	IMIN0	INTEGER*2	INTEGER*2
the argument			JMIN0	INTEGER*4	INTEGER*4
list; there		MIN1	IMIN1	REAL*4	INTEGER*2
must be at			JMIN1	REAL*4	INTEGER*4
least two		AMIN0	AIMIN0	INTEGER*2	REAL*4
arguments)			AJMIN0	INTEGER*4	REAL*4
Positive Difference	2	DIM	IIDIM	INTEGER*2	INTEGER*2
$a_1-(\min(a_1,a_2))$			JIDIM	INTEGER*4	INTEGER*4
			DIM	REAL*4	REAL*4
(returns the first			DDIM	REAL*8	REAL*8
argument minus			QDIM	REAL*16	REAL*16
the minimum of		IDIM	IIDIM	INTEGER*2	INTEGER*2
the two			JIDIM	INTEGER*4	INTEGER*4
arguments)					

[10]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.

Table D–3 (Cont.): VAX FORTRAN Intrinsic Functions

Functions	No. of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Remainder a_1-a_2*[a_1/a_2] (returns the remainder when the first argument is divided by the second)	2	MOD	IMOD JMOD AMOD DMOD QMOD	INTEGER*2 INTEGER*4 REAL*4 REAL*8 REAL*16	INTEGER*2 INTEGER*4 REAL*4 REAL*8 REAL*16
Transfer of Sign |a_1| Sign a_2	2	SIGN ISIGN	IISIGN JISIGN SIGN DSIGN QSIGN IISIGN JISIGN	INTEGER*2 INTEGER*4 REAL*4 REAL*8 REAL*16 INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4 REAL*4 REAL*8 REAL*16 INTEGER*2 INTEGER*4
Bitwise AND (performs a logical AND on corresponding bits)	2	IAND	IIAND JIAND	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
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

Table D–3 (Cont.): VAX FORTRAN Intrinsic Functions

Functions	No. of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Bit Extraction (extracts bits a_2 thru a_2+a_3-1 from a_1; see also MVBITS system subroutine)	3	IBITS	IIBITS JIBITS	INTEGER*2 INTEGER*4	INTEGER*4 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*4 INTEGER*4
Bitwise Circular Shift (circularly shifts rightmost a_3 bits of argument a_1 by a_2 places; bits in a_1 beyond the value specified by a_3 are unaffected)	3	ISHFTC	IISHFTC JISHFTC	INTEGER*2 INTEGER*4	INTEGER*4 INTEGER*4
Length (returns length of the character expression; see Chapter 2 for additional information on character functions)	1	—	LEN	CHARACTER	INTEGER*4

Table D–3 (Cont.): VAX FORTRAN Intrinsic Functions

Functions	No. of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Index (C1,C2) (returns the position of the substring c_2 in character expression c_1; see Chapter 2 for additional information on character functions)	2	—	INDEX	CHARACTER	INTEGER*4
Character (returns a character that has the ASCII value specified by the argument; see Chapter 2 for additional information on character functions)	1	—	CHAR	LOGICAL*1 INTEGER*2 INTEGER*4	CHARACTER
Nworkers (returns the total number of processes executing a routine)	0	—	NWORKERS	—	INTEGER*4
Sizeof (returns the number of bytes of storage used by the argument)	1	—	SIZEOF	Anything with a valid data type, except assumed-size arrays or passed-length characters	INTEGER*4

Table D–3 (Cont.): VAX FORTRAN Intrinsic Functions

Functions	No. of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
ASCII Value (returns the ASCII value of the argument; the argument must be a character expression that has a length of 1; see Chapter 2 for additional information on character functions)	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

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 as follows:

DATE	Returns a 9-byte string containing the ASCII representation of the current date in the form dd-mmm-yy.
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, in the form hh:mm:ss.
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 values are 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 takes the following form:

```
CALL DATE(buf)
```

buf

Is a 9-byte variable, array, array element, or character substring. The date is returned as a 9-byte ASCII character string taking the following form:

```
dd-mmm-yy
```

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 takes the following form:

```
CALL IDATE(i,j,k)
```

If the current date were October 9, 1988, the values of the integer variables upon return would be as follows:

```
i = 10
j = 9
k = 88
```

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 takes the following form:

```
CALL ERRSNS(fnum,rmssts,rmsstv,iunit,condval)
```

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 the *VAX FORTRAN User Manual*.

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
Is an integer variable or array element in which the RMS completion status code (STS) is stored, if the last error was an RMS I/O error.

rmsstv
Is an integer variable or array element in which the RMS status value (STV) is stored, if the last error was an RMS I/O error. This status value provides additional status information.

iunit
Is an integer variable or array element in which the logical unit number is stored, if the last error was an I/O error.

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 have an 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)]
```

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 takes the following form:

```
y = SECNDS(x)
```

y
Is set equal to the time in seconds since midnight, minus the user-supplied value of x.

The SECNDS function can perform elapsed-time computations. For example:

```
C     START OF TIMED SEQUENCE
      T1 = SECNDS(0.0)
C     CODE TO BE TIMED
      DELTA = SECNDS(T1)
```

DELTA gives the elapsed time.

The value of SECNDS is accurate to 0.01 second, which is the resolution of the system clock.

The time is computed from midnight. The SECNDS subroutine also produces correct results for time intervals that span midnight.

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 the following 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 takes the following form:

```
CALL TIME(buf)
```

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 having the following form:

```
hh:mm:ss
```

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 takes the following form:

```
y = RAN(i)
```

y
Is set equal to the value associated, by the function, with the argument i. The argument i is called the *seed*. It 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 = MOD (69069 * SEED + 1, 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.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 following:

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. For example, the integer -47 is represented by the following:

Binary pattern:	0...1010001
Bit position:	n...6543210 Where n is the number of bit positions in the numeric storage unit.

Notice that the value of bit position n is as follows:

1 — for a negative number
0 — for a non-negative number

All the high-order bits in 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 that operates on either of the two integer types and a specific form for each type. When you use the intrinsic functions that refer to bit positions or that 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. (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.

D.5.3 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 takes the following form:

```
CALL MVBITS(m,i,len,n,j)
```

m
Is an integer variable or array element that represents the source location (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 (the location to which a bit field is transferred).

j
Is an integer expression that identifies the starting position, within n, for the bits being transferred.

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.

Index

-
  See Subtraction or unary minus operator
!
  See Exclamation point
"
  See Quotation marks
$
  See Dollar sign
*
  See Asterisk
**
  See Exponentiation operator
+
  See Addition or unary plus operator
:
  See Colon
?
  See Question mark
/
  See Division operator
//
  See Concatenation operator

A

ABS intrinsic function • D–36
Absolute Value intrinsic function • D–36
ACCEPT statement • 7–47 to 7–48
ACCESS
  INQUIRE statement specifier • 9–25
  OPEN statement keyword • 9–4, 9–8
Access, shared
  SHARED keyword
    on OPEN statement • 9–4, 9–21
Access keys
  for indexed files
    specified in OPEN statement • 9–15
Access modes
  direct in OPEN statement keywords • 9–2
ACOSD intrinsic function • D–34
ACOS intrinsic function • D–34
Actual arguments
  aggregate field references used as • 2–37
  external procedure names used
    as • 4–21 to 4–22
  unsubscripted array used as • 2–29
Addition operator (+) • 2–42 to 2–44, 2–51
Adjustable arrays • 2–29
  in RECORD statements • 2–38
A field descriptor • 8–29 to 8–31
Aggregate assignment statement • 3–6
Aggregate field reference
  examples • 2–38 to 2–39
  format of • 2–36 to 2–37
Aggregate reference • 2–39 to 2–41
AIMAG intrinsic function • D–39
AINT intrinsic function • D–36
Allocation
  see File storage allocation
Alternate return argument • 6–9
AMAX0 intrinsic function • D–40
AMIN0 intrinsic function • D–40
.AND.
  See logical operators
ANINT intrinsic function • D–37

ANSI Standard
comparison with ISO Standard • 1–1
VAX FORTRAN extensions of • 1–1
APPEND
OPEN statement keyword value • 9–4
'APPEND'
OPEN statement keyword value • 9–8
Arc Cosine (degree) intrinsic function • D–34
Arc Cosine intrinsic function • D–34
Arc Sine (degree) intrinsic function • D–34
Arc Sine intrinsic function • D–34
Arc Tangent (degree) intrinsic function • D–35
Arc Tangent intrinsic function • D–35
Arguments
See also Actual arguments, Dummy arguments
aggregate field references as • 2–37
associating array elements with • 2–25
associating variables with • 2–22
bit function arguments • D–50 to D–52
external procedure names used as • 4–21 to 4–22
general description • 6–2 to 6–11
alternate return arguments • 6–9
arrays • 6–3 to 6–5
assumed-size arrays • 6–6
character arrays • 6–7
defaults for arguments passing • 6–9 to 6–11
Hollerith and character constants • 6–8
overview • 6–2
passed-length character arguments • 6–7
intrinsic function names used as • 4–23
use of built-in functions
argument list functions (%VAL, %REF, %DESCR) • 6–9 to 6–11
%LOC function • 6–11
Arithmetic assignment statement • 3–1 to 3–3
Arithmetic expressions • 2–42 to 2–46
compile-time • 4–27, 4–28
data type ranking • 2–45
in relational expressions • 2–48
operator precedence • 2–43 to 2–44
order of evaluation • 2–43 to 2–45
rules governing typing of • 2–45 to 2–46
Arithmetic IF statement • 5–16
Arithmetic operators
in expressions • 2–42 to 2–44, D–1
Array element
defining values of • 3–1
Array name
references to • 2–39 to 2–41
unsubscripted in a DATA statement • 4–6
Array references
in dimension bounds expressions • 2–26
without subscripts • 2–29
with subscripts • 2–27
Arrays
addressing character substrings in elements of • 2–30 to 2–31
adjustable • 2–29
arrangement of elements • 2–27
associating two or more • 2–25
assumed-size • 2–30
data types of • 2–1
data typing • 2–29
declarators • 2–25 to 2–26
defining in a COMMON statement • 4–4
defining with character type declarations • 4–11
definition of • 2–4
dimension bounds expressions • 2–25 to 2–26
dimensioning • 4–12
elements as scalar references • 2–39
establishing with subprogram references • 2–25
general description • 2–24 to 2–25
inability to dimension within CONTEXT_SHARED • 10–3
initializing with DATA statements • 4–5 to 4–8
in structure declaration blocks • 2–32
making equivalent • 4–14 to 4–16
references to in statements • 2–29
subscripts • 2–27
unsubscripted array names
usage restrictions • 2–29
ASCII character set • B–2
ASCII collating sequence functions • D–45
ASCII constants
assigned in DATA statements • 4–7
ASCII Value intrinsic function • D–45
ASIND intrinsic function • D–34
ASIN intrinsic function • D–34
Assigned GO TO statement • 5–14 to 5–15
establishing symbolic statement labels for • 3–7 to 3–8

Assignment statements
aggregate • 3–6
arithmetic • 3–1 to 3–3
character • 3–4 to 3–5
logical • 3–4
ASSIGN statement • 3–7 to 3–8
ASSOCIATEVARIABLE
OPEN statement keyword • 9–4, 9–8
Association
of arrays • 2–24
of variables • 2–22
Assumed-size arrays • 2–30
Asterisk (*)
comment line indicator • 1–8, 1–13
format specifier
in list-directed I/O • 7–2, 7–5
multiplication operator • 2–42 to 2–44, 2–51
upper bound of array • 2–25
Asterisk (*) length specifier
for dummy argument or function name • 4–11
in numeric type declarations • 4–9
ATAN2 intrinsic function • D–35
ATAND2D intrinsic function • D–35
ATAND intrinsic function • D–35
ATAN intrinsic function • D–35

B

BACKSPACE statement
See also REWIND statement
general description • 9–35
Binary operators
definition of • 2–43
Bit Clear intrinsic function • D–42
Bit Extraction intrinsic function • D–42
Bit field transfers
MVBITS subroutine • D–53
Bit functions
general information about • D–50 to D–52
Bit Set intrinsic function • D–42
Bit Test intrinsic function • D–42
Bitwise AND intrinsic function • D–41
Bitwise Circular Shift intrinsic function • D–43
Bitwise Complement intrinsic function • D–41
Bitwise Exclusive OR intrinsic function • D–41
Bitwise OR intrinsic function • D–41
Bitwise Shift intrinsic function • D–42
BLANK
INQUIRE statement specifier • 9–26
OPEN statement keyword • 9–4, 9–8
Blank common block • 4–3
Blank control editing
BN and BZ edit descriptors • 8–10 to 8–11
Blank lines • 1–8, 1–13
Block DATA statement • 4–2 to 4–3
BLOCK DATA statement
position after OPTIONS statement • 1–19
Block DATA subprogram • 4–2 to 4–3
BLOCK DATA subprogram
forcing linker to search libraries • 4–21
Block IF constructs • 5–17 to 5–24
Blocks
OPEN keywords affecting • 9–2
BLOCKSIZE
OPEN statement keyword • 9–4, 9–9
BN edit descriptor • 8–10
BTEST intrinsic function • D–42
BUFFERCOUNT
OPEN statement keyword • 9–4, 9–9
Built-in functions
argument list functions
%VAL, %REF, %DESCR • 6–9 to 6–11
%LOC function • 6–11
BYTE
as a data type • 2–2
as storage location • 2–3
declaring data type • 4–8 to 4–10
equivalence with LOGICAL*1 • 4–9
BZ edit descriptor • 8–11

C

C
to begin a compiler directive • 1–13
See also Compiler directives
to indicate a comment line • 1–8, 1–13
Calendar dates
subroutines for calculating
DATE and IDATE • D–46
CALL statement • 5–2 to 5–3
use of return args • 5–25, 5–27
use with ENTRY statement • 6–24
use with SUBROUTINE statement • 6–18, 6–20

CALL statement (cont'd.)
using names declared in INTRINSIC statement • 4–23
using names specified in EXTERNAL statement • 4–21
CARRIAGECONTROL
INQUIRE statement keyword • 9–26
OPEN statement keyword • 9–4, 9–10
Carriage control characters • 8–38 to 8–39
Carriage control editing • 8–38 to 8–39
CDD$TOP • 1–16
CDD records • 1–15
CDEC$ IDENT directive • 10–10
CDEC$ PSECT directive • 10–11 to 10–12
CDEC$ SUBTITLE directive • 10–13
CDEC$ TITLE directive • 10–13
CHARACTER
data type
definition • 2–1
representation in memory • C–11
storage requirement • 2–2 to 2–3
Character arguments
passed length • 2–23
Character assignment statement • 3–4 to 3–5
Character comparison library functions
LEN, INDEX, ICHAR, CHAR • 6–30 to 6–32
See also Lexical comparison functions
Character constants
as actual arguments • 6–8
assigned in DATA statements • 4–7
general description • 2–18 to 2–19
upper and lowercase letters in • 1–9
use of space within • 1–9
Character editing (A,H) • 8–29 to 8–32
character constants • 8–32
Character expressions • 2–47
compile-time • 4–27
in relational expressions • 2–48
CHARACTER FUNCTION statement • 6–15
Character intrinsic function • D–44
Character operators • 2–47
in expressions • D–1
Character relational intrinsic functions • D–45
Characters
See also uppercase characters, lowercase characters
declaring variables • 4–10
Characters (cont'd.)
defining values of • 3–1
in character and Hollerith constants • 1–9
printable/nonprintable • 1–9
Character set • 1–8
ASCII • B–2
FORTRAN • B–1
Radix–50 • B–4 to B–5
Character storage units • 2–2
Character substrings
addressing in variables, arrays, and array elements • 2–30 to 2–31
as scalar references • 2–39
definition • 2–4
making equivalent • 4–17 to 4–20
Character type declaration statement
general description • 4–10 to 4–12
CHAR function • 6–32
CHAR intrinsic function • D–44
CLOSE statement
general description • 9–23 to 9–24
CMPLX intrinsic function • D–39
Coding form • 1–10
Colon (:)
edit descriptor • 8–38
Columns
comment indicator position • 1–13
continuation indicator position • 1–14
in a line • 1–2
of a statement label field • 1–13
sequence number field position • 1–15
statement field position • 1–14
Comment • 1–7 to 1–8
allowable characters in • 1–9
in a parallel DO-loop • 10–4
Comment line indicator
D in column 1 • 1–14
general description • 1–7
common block
names in COMMON statement • 4–4
Common block
COMMON and EQUIVALENCE interaction • 4–20
default attributes of • 10–12t
establishing order of contents • 4–4
initializing values in • 4–2 to 4–3

Common block
in parallel processing
specifying private default shareability • 10–7
specifying private shareability • 10–6
specifying shared default shareability • 10–8
specifying shared shareability • 10–7
modifying attributes of • 10–11
modifying shareability in parallel DO-loop • 10–2 to 10–7
Common Data Dictionary
See DICTIONARY statement
Common logarithm intrinsic function • D–33
COMMON statement
See also Common block
establishing arrays with • 2–24, 2–25
establishing variables with • 2–22
general description • 4–3 to 4–5
interaction with EQUIVALENCE • 4–20
using unsubscripted array names • 2–29
Compilation options
overriding FORTRAN command options • 1–18 to 1–19
Compiler directives • 10–1 to 10–13
Compile-time constant expressions • 4–27 to 4–28
COMPLEX
declaring data type • 4–8 to 4–10
COMPLEX*16
constants • 2–14
data type
representation in memory • C–8 to C–10
storage requirement • 2–2 to 2–3
D_floating and G_floating implementations • 2–4
COMPLEX*8
constants • 2–13 to 2–14
data type
definition • 2–1
representation in memory • C–8
storage requirement • 2–2 to 2–3
Complex Conjugate intrinsic function • D–39
Complex data editing • 8–25
Complex entities
in relational expressions • 2–48
Complex from Two Arguments function
Complex from Two Arguments function (cont'd.)
See CMPLX intrinsic function, DCMPLX intrinsic function
Complex numbers
intrinsic functions that operate on • D–39
intrinsic functions to operate on • 6–26
Computed GO TO statement • 5–13 to 5–14
Concatenation operator (//) • 2–47, 2–51
CONJG intrinsic function • D–39
Connections, logical
to logical I/O units
explicitly by means of OPEN • 9–22
Constants
as scalar references • 2–39
assigning symbolic names with PARAMETER statement • 4–26 to 4–28
character • 2–18
complex • 2–13 to 2–15
data types of • 2–1
definition • 2–4
hexadecimal • 2–15 to 2–18
Hollerith • 2–19 to 2–21
initialized by a PARAMETER statement • 2–46
integer • 2–5
logical • 2–18
octal • 2–15 to 2–18
Radix–50 • B–4
real • 2–8 to 2–13
specifying data types of • 2–5 to 2–21
using a symbolic name • 4–27
CONTEXT_SHARED directive • 10–3
CONTEXT_SHARED_ALL directive • 10–3
Continuation indicator field • 1–10, 1–14
Continuation line • 1–2
effect on statement labels • 1–13
in debug source statements • 1–14
indicator in source code • 1–14
in included files • 1–17
use of in compiler directive • 10–1
CONTINUE statement • 5–3
Control characters • 1–9
Control list parameters
general description • 7–2 to 7–3
Control statements • 5–1 to 5–28
See also CALL, RETURN
in parallel DO-loop • 10–4
Control transfer
FORTRAN control statements • 5–1 to 5–28

Control transfer (cont'd.)
with DO-loops • 5–8
Conversion
of data types in arithmetic assignments • 3–3t
with FORMAT statements • 8–1
Conversion to COMPLEX*16 function • D–39
Conversion to COMPLEX*8 function • D–39
Conversion to REAL*16 function • D–38
Conversion to REAL*4 function • D–37
Conversion to REAL*8 function • D–38
COSD intrinsic function • D–34
COSH intrinsic function • D–35
Cosine (degree) intrinsic function • D–34
Cosine intrinsic function • D–34
COS intrinsic function • D–34
CPAR$ CONTEXT_SHARED directive • 10–3
CPAR$ CONTEXT_SHARED_ALL directive • 10–3
CPAR$ DO_PARALLEL directive • 10–4 to 10–5
CPAR$ LOCKOFF directive • 10–5 to 10–6
CPAR$ LOCKON directive • 10–5 to 10–6
used to resolve data dependences • 10–5
CPAR$ PRIVATE directive • 10–6 to 10–7
CPAR$ PRIVATE_ALL directive • 10–7
CPAR$ SHARED directive • 10–7
CPAR$ SHARED_ALL directive • 10–8
Critical region
in a parallel DO-loop • 10–5

D

D
in REAL*8 constants • 2–11
D_floating data implementation
representation in memory
COMPLEX*16 • C–8 to C–9
REAL*8 • C–3, C–5
with COMPLEX*16 data type • 2–4, 2–14
with REAL*8 data type • 2–4, 2–10
/D_LINES qualifier • 1–14
Data
as stored in memory
by VAX FORTRAN • C–1 to C–12
editing
with FORMAT statements • 8–1
retaining after END or RETURN • 4–30 to 4–31
Data dependences
in a parallel DO-loop • 10–5
Data items
as scalar references • 2–39
defining values for • 3–1
definition • 2–4
identified by symbolic names • 1–7
DATA statement
general description • 4–5 to 4–8
to define arrays and elements • 2–25
using unsubscripted array names • 2–29
Data type declaration statement
See also Type declaration statement
character type declarations • 4–10 to 4–12
numeric type declarations • 4–8 to 4–10
use to establish arrays • 2–24
using unsubscripted array names • 2–29
Data types
ability of entities to have • 1–7
conversion in arithmetic assignment
statements • 3–3t
declaration within structures • 4–34
default data types of undeclared names • 4–22
definition of different types • 2–1 to 2–2
establishing in arithmetic
expressions • 2–45 to 2–46
length specifiers • 2–2
rank in arithmetic expressions • 2–45
specifying for arrays • 2–28
specifying for constants • 2–5 to 2–21
specifying for variables • 2–22, 2–23 to 2–24
Dates, calendar
subroutines for calculating
DATE and IDATE • D–46
DATE subroutine • D–46
DBLE intrinsic function • D–38
DCMPLX intrinsic function • D–39
D debugging statement indicator
in column 1 • 1–14
Debugging statements • 1–14
Declarators, array • 2–25 to 2–26
DECODE statement • A–1 to A–3
DEFAULTFILE
INQUIRE statement keyword • 9–25
OPEN statement keyword • 9–4, 9–10
Defaults
argument passing defaults • 6–10
data type of undeclared names • 4–22
field descriptors vs. I/O list elements • 8–33

DEFINE FILE statement • A–3 to A–4
DELETE
  file description • 9–23
DELETE statement
  general description • 9–36 to 9–37
Delimiting periods
  of logical values • 2–50
  of relational values • 2–48
%DESCR built-in function • 6–10
D field descriptor • 8–21
  in complex data editing • 8–25
DFLOAT intrinsic function • D–38
DICTIONARY statement • 1–15 to 1–16
DIMAG intrinsic function • D–39
Dimension bounds expressions
  See Arrays
Dimensions
  array limits • 2–24
  declaring of an array • 2–25 to 2–26
DIMENSION statement
  general description • 4–12 to 4–13
  use to establish arrays • 2–24
DIM intrinsic function • D–40
DIRECT
  INQUIRE statement specifier • 9–27
  OPEN statement keyword value • 9–4
Direct access FIND statements • A–5
Direct access mode
  OPEN statement keywords • 9–2
Direct access READ statements • 7–26 to 7–27
Direct access WRITE statements • 7–39 to 7–40
Directives
  See Compiler directives
'DIRECT'
  OPEN statement keyword value • 9–8
DISP
  CLOSE statement keyword • 9–23
DISPOSE
  CLOSE statement keyword • 9–23
  OPEN statement keyword • 9–4, 9–11
Division operator (/) • 2–42 to 2–44, 2–51
Dollar sign ($)
  delimiter for namelist record • 7–20
  edit descriptor • 8–37 to 8–38
  in a symbolic name • 1–6
DO-loop
  See also Parallel DO-loop
DO-loop (cont'd.)
  transferring control • 5–8
DO statements • 5–3 to 5–12
  indexed • 5–4 to 5–9
  pretested indefinite
    DO WHILE • 5–9 to 5–11
DOUBLE COMPLEX
  declaring data type • 4–8 to 4–10
DOUBLE PRECISION
  declaring data type • 4–8 to 4–10
DO WHILE statement • 5–9 to 5–11
DO_PARALLEL directive • 10–4 to 10–5
DPROD intrinsic function • D–39
DREAL intrinsic function • D–39
Dummy arguments
  aggregate field references used as • 2–37
  inability to declare in
    CONTEXT_SHARED • 10–3
  unsubscribed array names used as • 2–29
  using asterisk (*) length specifier • 4–11

E

Edit descriptors
  summary • 8–6 to 8–7
E field descriptor • 8–19
  in complex data editing • 8–25
ELSE IF THEN statement
  block IF constructs • 5–17 to 5–24
ELSE statement
  block IF constructs • 5–17 to 5–24
ENCODE statement • A–1 to A–3
END DO statement • 5–11 to 5–12
ENDFILE statement • 9–35 to 9–36
END IF statement
  block IF constructs • 5–17 to 5–24
END MAP statement • 4–39
End-of-file condition
  reporting with IOSTAT value • 7–9
  transferring control with END specifier • 7–10
End-of-file record
  ENDFILE statement • 9–35 to 9–36
END specifier
  in I/O statements • 7–10
END statement
  general description • 5–12
  with BLOCK DATA statement • 4–3

END statement (cont'd.)
with FUNCTION statement • 6–15
with SUBROUTINE statement • 6–18
END UNION statement • 4–39
ENTRY statement • 6–21 to 6–24
use with FUNCTION statement • 6–17
use with SUBROUTINE statement • 6–19
using unsubscripted array names • 2–29
.EQ.
See relational operators
EQUIVALENCE statement
associating arrays with • 2–25
associating variables with • 2–22
contrasted with union declaration • 4–40
general description • 4–13 to 4–20
interaction with COMMON • 4–20
use of unsubscripted arrays with • 2–22, 2–25
using unsubscripted array names • 2–29
.EQV.
See logical operators
ERR
BACKSPACE statement keyword • 9–35
CLOSE statement keyword • 9–23
DELETE statement keyword • 9–36
ENDFILE statement keyword • 9–35
I/O statement specifier • 7–10
INQUIRE statement specifier • 9–27
OPEN statement keyword • 9–4, 9–12
REWIND statement keyword • 9–34
UNLOCK statement keyword • 9–38
Error condition
during I/O • 7–9, 7–10
Error handling
subroutine for obtaining error information
ERRSNS subroutine • D–47
user controls in I/O statements
ERR, END, and IOSTAT specifiers • 7–9, 7–10
ERRSNS subroutine • D–47
Exclamation point (!)
comment indicator • 1–8, 1–13
Executable statements • 1–2, 1–4
Execution, program
temporarily suspending
(PAUSE) • 5–24 to 5–25
terminating
EXIT • D–48
STOP • 5–27
EXIST
INQUIRE statement specifier • 9–27
EXIT system subroutine • D–48
EXP intrinsic function • D–33
Explicit formatting
I/O statement specifier • 7–4 to 7–5
Exponential intrinsic function • D–33
Exponentiation operator (**) • 2–42 to 2–44, 2–51
Exponents
in REAL*4 constants • 2–8
in REAL*8 constants • 2–10
in REAL*16 constants • 2–12
Expressions, FORTRAN
See also Arithmetic expressions, Character expressions, Logical expressions, Relational expressions
as scalar references • 2–39
compile-time constant expressions • 4–27 to 4–28
data types of • 2–1
definition of • 2–42
expression operators
summary of • D–1 to D–2
variable FORMAT • 8–9 to 8–10
Extended ranges, DO-loop • 5–9
EXTENDSIZE
OPEN statement keyword • 9–4, 9–12
External field separators • 8–40 to 8–41
External procedure names
as arguments • 4–21 to 4–22
duplicating intrinsic function names • 4–21
External procedures
invoking with CALL • 5–2
unsubscripted array names as dummy arguments • 2–29
EXTERNAL statement • 4–21 to 4–22
/NOF77 implementation • A–8 to A–9
to search object libraries for block data subprograms • 4–3

F

F_floating data implementation
representation in memory
COMPLEX*8 • C–8
REAL*4 • C–4

F field descriptor • 8–17
in complex data editing • 8–25
Field
in FORTRAN source code • 1–9 to 1–15
Field declarations
allowable entities • 4–35
Field descriptors
defaults for I/O list elements • 8–33
summary • 8–6 to 8–7
Field namelist • 4–34
Field names • 4–34
in structure declarations • 4–38
Field references
See Record and field references
Fields
definition in structure declarations • 4–33 to 4–38
initialization of unnamed fields • 2–33
Fields, unnamed
use of %FILL • 2–33
Field separators, external • 8–40 to 8–41
File
combining at compilation • 1–17 to 1–18
deleting records from
DELETE statement • 9–36 to 9–37
disposition
CLOSE statement keywords • 9–23
INCLUDE files • 1–17 to 1–18
processing options
OPEN statement keywords • 9–2
properties, inquiring about
INQUIRE statement • 9–24 to 9–33
providing a listing header for • 10–13
record description options, I/O
OPEN statement keywords • 9–2
repositioning
BACKSPACE statement • 9–35
repositioning with REWIND statement • 9–34
status options
OPEN statement keywords • 9–2
FILE
INQUIRE statement keyword • 9–25
OPEN statement keyword • 9–4, 9–13
File-handling commands
BACKSPACE statement • 9–35
CLOSE statement • 9–23 to 9–24
INQUIRE statement • 9–24 to 9–33
OPEN statement • 9–2 to 9–23
File-handling commands (cont'd.)
REWIND statement • 9–34
File sharing
SHARED keyword (OPEN statement) • 9–4, 9–21
File specifications
OPEN statement keywords • 9–2
File status
CLOSE statement keywords • 9–23
OPEN statement keywords
DISPOSE • 9–4, 9–11
STATUS or TYPE • 9–4, 9–22
File storage allocation
OPEN statement keywords • 9–2
%FILL • 2–33, 4–34, 4–38
FIND statement • A–5
Fixed-format • 1–10 to 1–11
Fixed-length records
RECORDTYPE keyword (OPEN statement) • 9–4, 9–20 to 9–21
Fix intrinsic function • D–38
Floating-point data types
emulation of • 2–4
representation in memory • C–3 to C–10
FLOAT intrinsic function • D–38
FMT format specifier
in I/O statements • 7–4
FORM
INQUIRE statement specifier • 9–28
OPEN statement keyword • 9–4, 9–13
Format
coding with fixed format • 1–10 to 1–11
coding with tab format • 1–11 to 1–13
Formats
passed length • 2–2
run-time • 8–41 to 8–42
Format specification separators • 8–39 to 8–40
Format specifier
control list parameter
in I/O statements • 7–4
FORMAT statements
arithmetic expressions in • 8–9 to 8–10
description of use • 8–1
external field separators • 8–40 to 8–41
field and edit descriptors
additional editing operations (Q,$,:) • 8–36 to 8–38
blank control editing, (BN,BZ) • 8–10

FORMAT statements
field and edit descriptors (cont'd.)
character editing (A,H) • 8–29 to 8–32
counts, repeat • 8–8
integer editing (I,O,Z) • 8–12 to 8–17
logical editing (L) • 8–28
positional editing (X,T,TL,TR) • 8–34 to 8–36
real editing (F,E,D,G) • 8–17 to 8–25
scale factor editing (P) • 8–25 to 8–27
sign control editing (SP,SS,S) • 8–11
summary of • 8–6 to 8–7
use of character constants • 8–32
format expressions, variable • 8–9 to 8–10
format specification
separators • 8–39 to 8–40
general rules • 8–2 to 8–3
I/O lists, interaction with • 8–42 to 8–46
input rules • 8–3
output rules • 8–3 to 8–4
run-time formats • 8–41 to 8–42
syntax • 8–4 to 8–6
FORMATTED
INQUIRE statement specifier • 9–28
Formatted I/O statements
ACCEPT statement • 7–47
establishing symbolic statement labels
for • 3–7 to 3–8
PRINT statement • 7–48 to 7–49
READ statements
direct access • 7–26
indexed • 7–28, 7–29
internal • 7–31
sequential • 7–15, 7–16
REWRITE statement • 7–45, 7–46
TYPE statement • 7–48 to 7–49
WRITE statements
direct access • 7–39, 7–40
indexed • 7–41, 7–42
internal • 7–43, 7–44
sequential • 7–33, 7–34 to 7–35
FORTRAN-66
VAX FORTRAN support of • 1–1
FORTRAN–77
VAX FORTRAN extensions of • 1–1
FORTRAN character set • B–1
FORTRAN command (DCL)
/D_LINES • 1–14
FORTRAN command (DCL) (cont'd.)
overriding • 1–15, 1–18 to 1–19
FORTRAN data representation in memory • C–1 to C–12
FORTRAN statements • 1–2 to 1–5
assignment statements • 3–1 to 3–8
control statements • 5–1 to 5–28
I/O statements • 7–1 to 7–49
I/O statements, auxiliary • 9–1 to 9–38
language summary (alphabetic) • D–2 to D–32
scalar field references in • 2–37
specification statements • 4–1 to 4–42
supplemental statements
supported to maintain non-VAX FORTRAN compatibility • A–1 to A–9
Function name
using asterisk (*) length specifier • 4–11
Function references
data types of • 2–1
general description • 6–16 to 6–18
types of references to intrinsic functions
specific and generic • 6–25 to 6–30
Functions
See Built-in functions
See Intrinsic functions, system supplied
See Lexical comparison library functions
See Statement functions
FUNCTION statement • 6–15 to 6–18
logical and numeric functions • 6–15
position after OPTIONS statement • 1–19
using unsubscripted array names • 2–29
Function subprograms • 6–15 to 6–18

G

G_floating data implementation
representation in memory
COMPLEX*16 • C–10
REAL*8 • C–6
with COMPLEX*16 data type • 2–4, 2–14
with REAL*8 data type • 2–4, 2–10
.GE.
See relational operators
General directives • 10–10 to 10–13
ordering with statements • 1–3f
Generic references

Generic references (cont'd.)
to intrinsic function names • 6–26 to 6–27, 6–28
G field descriptor • 8–22
in complex data editing • 8–25
GO TO statement
establishing symbolic statement labels for assigned • 3–7 to 3–8
GO TO statements • 5–12 to 5–15
Group repeat counts
in FORMAT statements • 8–8
.GT.
See relational operators

H

H_floating data implementation
representation in memory
REAL*16 • C–7
Hexadecimal constants • 2–5, 2–15 to 2–18
assigned in DATA statement • 4–7
data type assignments • 2–16 to 2–17
H field descriptor • 8–32
Hollerith constants • 2–5, 2–19 to 2–21
assigned in DATA statements • 4–7
upper and lowercase letters in • 1–9
use of space within • 1–9
Hollerith Constants
representation in memory • C–11
Hyperbolic Cosine intrinsic function • D–35
Hyperbolic Sine intrinsic function • D–35
Hyperbolic Tangent intrinsic function • D–35

I

I
field descriptor • 8–12 to 8–14
I/O, iterative
See iterative I/O
I/O statement components
control list parameters • 7–1 to 7–3
format specifier • 7–4
I/O status specifier • 7–9
internal file specifier • 7–4
key-field value specifier • 7–6
key-of-reference specifier • 7–9
logical unit specifier • 7–3
I/O statement components
control list parameters (cont'd.)
namelist specifier • 7–5
record specifier • 7–6
rules for specifying • 7–3
transfer-of-control specifier • 7–10
I/O list parameter • 7–11 to 7–14
implied-DO lists • 7–13 to 7–14
interaction with format controls • 8–42 to 8–46
simple list elements • 7–12
I/O statements
See also I/O statement components, ACCEPT, FORMAT, OPEN, PRINT, READ, REWRITE, TYPE, WRITE
in parallel DO-loop • 10–4
list of • 7–1
OPEN statement interdependencies
logical unit specifier • 7–4
specifiers
See I/O statement components
using unsubscripted array names • 2–29
I/O status specifier
control list parameter
in I/O statements • 7–9
IABS intrinsic value • D–36
IAND intrinsic function • D–41
IBCLR intrinsic function • D–42
IBITS intrinsic function • D–42
IBTSET intrinsic function • D–42
ICHAR function • 6–31
ICHAR intrinsic function • D–45
IDATE subroutine • D–47
IDENT directive • 10–10
IDIMintrinsic function • D–40
IDINT intrinsic function • D–36
IDNINT intrinsic function • D–37
IEOR intrinsic function • D–41
IFIX intrinsic function • D–38
IF statements • 5–15 to 5–24
general descriptions
arithmetic IF • 5–16
block IF • 5–17 to 5–24
logical IF • 5–17
IF THEN statement
block IF constructs • 5–17 to 5–24
Imaginary Part of Complex function • D–39

IMPLICIT NONE statement • 4–23
IMPLICIT statement
  effect of /WARNINGS option • 4–23
  general description • 4–22 to 4–23
  using to type variables • 2–23
Implied-DO list
  See Iterative I/O
Implied-DO variables
  initializing with statements • 4–5 to 4–8
INCLUDE statement • 1–17 to 1–18
Indefinite DO statement, pretested
  DO WHILE • 5–9 to 5–11
Indexed DO statement • 5–4 to 5–9
Indexed files
  freeing locked records • 9–38
Indexed I/O statements
  READ statements • 7–28 to 7–30
  WRITE statements • 7–41 to 7–43
Indexed organization files
  access keys
    specifier in OPEN statement • 9–15
  deleting records from
    DELETE statement • 9–36 to 9–37
Indexed WRITE statements • 7–41 to 7–43
INDEX function • 6–31
INDEX intrinsic function • D–43
INITIALSIZE
  OPEN statement keyword • 9–4, 9–14
INQUIRE statement
  general description • 9–24 to 9–33
Integer
  constants
    in REAL*4 constants • 2–8
    octal notation • A–7
  data type
    definition • 2–1
    representation in memory • C–1, C–2
    storage requirements • 2–2 to 2–3
  declaring data type • 4–8 to 4–10
  default data type of undeclared names • 4–22
Integer constants
  in COMPLEX*8 constants • 2–13
  in REAL*8 constants • 2–10
  in REAL*16 constants • 2–12, 2–14
  used to assign values • 2–6
Integer data type
  See also Constants
Integer editing (I,O,Z) • 8–12 to 8–17
Internal file specifier
  control list parameter
    in I/O statements • 7–4
Internal I/O statements
  ENCODE and DECODE statements •
    A–1 to A–3
  READ statements • 7–31 to 7–32
  WRITE statements • 7–43 to 7–44
Internal WRITE statements • 7–43 to 7–44
INT intrinsic function • D–36
Intrinsic functions, system-supplied
  character comparison functions •
    6–30 to 6–32
  complete list of • D–32 to D–45
  description of types • 6–2
  external procedures of same name • 4–21
  lexical comparison functions • 6–32 to 6–33
  names used as arguments • 4–23
  references, generic • 6–26 to 6–27,
    6–28 to 6–30
  references, specific • 6–25, 6–28 to 6–30
INTRINSIC statement
  general description • 4–23 to 4–24
IOR intrinsic function • D–41
IOSTAT
  BACKSPACE statement keyword • 9–35
  CLOSE statement keyword • 9–23
  DELETE statement keyword • 9–36
  ENDFILE statement keyword • 9–35
  INQUIRE statement specifier • 9–28
  OPEN statement keyword • 9–4, 9–14
  REWIND statement keyword • 9–34
  specifier in I/O statements • 7–9
  UNLOCK statement keyword • 9–38
IQINT intrinsic function • D–36
IQNINT intrinsic function • D–37
ISHFTC intrinsic function • D–43
ISHFT intrinsic function • D–42
ISIGN intrinsic function • D–41
ISO Standard
  comparison with ANSI Standard • 1–1
Iterative I/O
  implied-DO list • 7–13 to 7–14
  iterative count controls
    indexed DO statement • 5–4 to 5–9
Iterative processing controls
  See DO statements

K

KEEP
  file disposition • 9–23
KEY
  key-field value specifier
    in I/O statements • 7–6
  OPEN statement keyword • 9–4, 9–15
KEYED
  INQUIRE statement specifier • 9–29
  OPEN statement keyword value • 9–4
'KEYED'
  OPEN statement keyword value • 9–8
Key-field value specifier
  control list parameter
    in I/O statements • 7–6
KEYID specifier
  see key-of-reference specifier • 7–9
Key-of-reference specifier
  control list parameter
    in I/O statements • 7–9
Keys, access
  specified in OPEN statement • 9–15
KEYx specifier (KEY, KEYEQ, KEYGE, KEYGT, KEYNXTNE, KEYNXT)
  See key-field value specifier

L

Labels
  See statement labels
.LE.
  See relational operators
L edit descriptor • 8–28
LEN function • 6–30
Length
  effect of /EXTEND_SOURCE on sequence number field • 1–15
  specifier in data type declarations • 2–2
Length intrinsic function • D–43
LEN intrinsic function • D–43
Lexical comparison library functions
  LLT, LLE, LGT, LGE • 6–32 to 6–33
<LF> control character • 1–9
LGE function • 6–32 to 6–33
LGT function • 6–32 to 6–33
Library functions, system-supplied
  algorithms used in • D–32
Line
  as a physical section of statements • 1–2
  blank • 1–8, 1–13
  characters embedded in • 1–9
  continuation indicator field • 1–14
  entering with fixed format • 1–10 to 1–11
  entering with tab format • 1–11 to 1–13
  format of statement label field • 1–13
  sequence number field • 1–15
  statement field • 1–14
Linefeed control character • 1–9
List-directed formatting
  I/O statement specifier • 7–4
List-directed I/O statements
  ACCEPT statement • 7–47
  READ statements
    internal READ • 7–31
    sequential READ • 7–15, 7–17 to 7–19
  WRITE statements
    internal WRITE • 7–43, 7–44
    sequential WRITE • 7–33, 7–35 to 7–37
List elements, simple
  I/O list parameter
    in I/O statements • 7–12
Listing
  See Source listing
Listing header
  providing for a file • 10–13
/LIST qualifier
  to enable TITLE and SUBTITLE compiler directives • 10–13
Lists, implied-DO
  in DATA statements • 4–5
LLE function • 6–32 to 6–33
LLT function • 6–32 to 6–33
%LOC built-in function • 6–11
Locked records
  freeing locked records • 9–38
LOCKOFF directive • 10–5 to 10–6
LOCKON directive • 10–5 to 10–6
  used to resolve data dependences • 10–5
Lock variable • 10–5
LOG10 intrinsic function • D–33

Logical
See also Arrays, Constants, Data types, Logical values, Variables
constants
representation in memory • C–2
storage requirement • 2–2 to 2–3
data type
definition • 2–2
relationship to BYTE data type • 4–9
LOGICAL
declaring data type • 4–8 to 4–10
LOGICAL*n
See Logical
Logical assignment statement • 3–4
Logical constants • 2–18
Logical editing (L) • 8–28
Logical elements
See Logical expressions
Logical expressions • 2–49 to 2–52
compile-time • 4–27
evaluation of subexpressions • 2–51
order of evaluation • 2–51 to 2–52
Logical functions • 6–15
Logical I/O units
CLOSE statement options • 9–23
connection method
explicitly by means of OPEN • 9–22
defining logical unit numbers
DEFINE FILE statement • A–3 to A–4
inquiring about properties
INQUIRE statement • 9–24 to 9–33
OPEN statement options • 9–2
Logical IF statement • 5–17
Logical operations
data types that result from • 2–50
Logical operators • 2–50t
in expressions • D–2
Logical scalar memory reference
See Scalar memory reference
Logical unit specifier
control list parameter
in I/O statements • 7–3
Logical values
representation in memory • C–2 to C–3
LOG intrinsic function • D–33
Loops, DO
DO statements • 5–3 to 5–12
Lowercase characters • 1–8
affect on compiler • 1–9
in character and Hollerith constants • 1–9
.LT.
See relational operators

M

Main program
as a program unit • 1–2
Map declaration
general description • 4–38 to 4–41
to establish variables • 2–22
use to establish arrays • 2–25
Mapped field declarations • 2–33
MAP statement • 4–38, 4–39
Mathematical functions, intrinsic • D–32 to D–45
MAX0 intrinsic function • D–40
MAX1 intrinsic function • D–40
Maximum intrinsic function • D–40
MAX intrinsic function • D–40
MAXREC
OPEN statement keyword • 9–4, 9–16
Memory diagrams
of structured records • 2–34 to 2–36
Messages
sending to terminal
See PAUSE statement
MIN0 intrinsic function • D–40
MIN1 intrinsic function • D–40
Minimum intrinsic function • D–40
MIN intrinsic function • D–40
Minus operator (–) • 2–42 to 2–44, 2–51
MOD intrinsic function • D–41
Multiplication operator (*) • 2–42 to 2–44, 2–51
MVBITS subroutine • D–53

N

Name
See also symbolic names, entry names
NAME
INQUIRE statement specifier • 9–29
OPEN statement keyword • 9–4, 9–17
Name, structure
using • 4–33

NAMED
INQUIRE statement specifier • 9–30
Named common blocks
establishing order of contents • 4–4
initializing values in • 4–2 to 4–3
Namelist-directed I/O statements
See also NAMELIST statement
ACCEPT statement • 7–47
sequential READ statement • 7–15, 7–20 to 7–25
sequential WRITE statement • 7–33, 7–37 to 7–38
Namelist specifier
control list parameter
in I/O statements • 7–5
NAMELIST statement
general description • 4–24 to 4–25
using unsubscripted array names • 2–29
Names, symbolic
See Symbolic names
Natural Logarithm intrinsic function • D–33
.NE.
See relational operators
Nearest Integer intrinsic function • D–37
.NEQV.
See logical operators
Nested block IF constructs • 5–23 to 5–24
Nested DO loops • 5–7 to 5–8
Nested structured declarations
See substructure declarations
Nesting
structure declarations • 4–33, 4–38
<NEWLINE> control character • 1–9
NEXTREC
INQUIRE statement specifier • 9–30
NINT intrinsic function • D–37
NML specifier
in I/O statements • 7–5
/NOF77 qualifier
effect on Do-loops • 5–5
/NOLIST qualifier
in the DICTIONARY statement • 1–16
in the INCLUDE statement • 1–17
Nonexecutable statements • 1–2
Nonprintable characters • 1–9
/NOOBJECT qualifier
disabling IDENT directive • 10–10
NOSPANBLOCKS
OPEN statement keyword • 9–4, 9–17
.NOT.
See logical operators
NOT intrinsic function • D–41
NUMBER
INQUIRE statement specifier • 9–30
Number, sequence • 1–15
Numerals • 1–8
Numeric functions • 6–15
Numeric scalar memory reference
See Scalar memory reference
Numeric storage unit • 2–2
Numeric type declarations
general description • 4–8 to 4–10
NWORKERS intrinsic function • D–44

O

O
field descriptor • 8–14
Object libraries
searching for block data subprograms • 4–3
Object module
labeling and identifying with compiler directive • 10–10
Octal constants • 2–5, 2–15 to 2–18
assigned in DATA statement • 4–7
data type assignments • 2–16 to 2–17
Octal notation (")
for integer constants • A–7
Octal values
I/O transfers
by O field descriptor • 8–14
OPENED
INQUIRE statement specifier • 9–31
OPEN statement
general description • 9–2 to 9–23
I/O statement interdependencies
logical unit specifier • 7–4
Operators
See also arithmetic operators, relational operators
expression operators
summary of • D–1 to D–2
logical • 2–50t

Operators (cont'd.)
precedence in arithmetic expressions • 2–43 to 2–44
precedence in relational expressions • 2–49
used in logical expressions • 2–51
Optimization
effect of VOLATILE statement • 4–41
OPTIONS statement • 1–3, 1–18 to 1–19
.OR.
See logical operators
Order
required of statements • 1–3t
ORGANIZATION
INQUIRE statement specifier • 9–31
OPEN statement keyword • 9–4, 9–17

P

Parallel directives • 10–2 to 10–9
See also NWORKERS function
enabling • 10–2
examples of • 10–8 to 10–9
ordering with statements • 1–3t
Parallel DO-loop • 10–4 to 10–9
/PARALLEL qualifier • 10–2
PARAMETER statement
alternate version of • A–6
defining constants in arithmetic expressions • 2–46
general description • 4–26 to 4–28
in structure declaration block • 4–32
Parentheses
effect in arithmetic expressions • 2–44 to 2–45
effect in character expressions • 2–47
in logical expression • 2–51
Passed-length character arguments • 2–23
Passed-length format, *(*)
for dummy arguments or character functions • 2–2
Pathnames (CDD) • 1–16
PAUSE statement • 5–24 to 5–25
in parallel DO-loop • 10–4
PDP–11 FORTRAN–77
source programs on a VAX FORTRAN compiler • 1–2
P edit descriptor • 8–25 to 8–27
Periods
delimiting logical values • 2–50
delimiting relational values • 2–48
Plus operator (+) • 2–42 to 2–44, 2–51
Positional editing (X,T,TL,TR) • 8–34 to 8–36
Positive Difference intrinsic function • D–40
Precedence, operator
effect of parentheses • 2–44 to 2–45
in relational expressions • 2–49
within arithmetic expressions • 2–43 to 2–44
Pretested indefinite DO statement
DO WHILE • 5–9 to 5–11
PRINT
file disposition • 9–23
PRINT statement • 7–48 to 7–49
PRIVATE directive • 10–6 to 10–7
Private entities
using SAVE statement for • 10–7
PRIVATE_ALL directive • 10–7
Procedure
See subprogram
Program
See program unit
Program execution
temporarily suspending (PAUSE) • 5–24 to 5–25
terminating
EXIT • D–48
STOP • 5–27
PROGRAM statement
general description • 4–28 to 4–29
position after OPTIONS statement • 1–19
Program unit
assigning a name to main program unit • 4–28 to 4–29
block data subprogram • 4–2
definition of • 1–2
PSECT directive • 10–11 to 10–12

Q

Q
edit descriptor • 8–37
in REAL*16 constants • 2–12
QEXT intrinsic function • D–38
QFLOAT intrinsic function • D–38

Question mark (?)
namelist prompt • 7–22
Quotation marks (")
octal notation for integer constants • A–7

R

Radix–50
constants and character sets • B–4 to B–5
Random number generator
RAN function • D–50
RAN function • D–50
READONLY
OPEN statement keyword • 9–4, 9–18
READ statements
direct access READ • 7–26 to 7–27
formatted • 7–26
unformatted • 7–26, 7–27
indexed READ • 7–28 to 7–30
formatted • 7–28, 7–29
unformatted • 7–28, 7–30
internal READ • 7–31 to 7–32
formatted • 7–32
list-directed • 7–31, 7–32
relationship to DECODE statement • A–1, A–2
sequential READ • 7–15 to 7–25
formatted • 7–15, 7–16
list-directed • 7–15, 7–17 to 7–19
namelist-directed • 7–15, 7–20 to 7–25
unformatted • 7–15, 7–25
REAL
See REAL*4
declaring data type • 4–8 to 4–10
REAL*16
See also Arrays, Constants, Data types, Variables
constants • 2–12 to 2–13
data type
definition • 2–1
representation in memory • C–4, C–7
storage requirements • 2–2 to 2–3
REAL*16 float intrinsic function • D–38
REAL*4
See also Arrays, Constants, Data types, Variables
constants • 2–8 to 2–10
REAL*4
data type
definition • 2–1
representation in memory • C–4
storage requirements • 2–2 to 2–3
default data type of undeclared names • 4–22
REAL*8
See also Arrays, Constants, Data types, Variables
constants • 2–10 to 2–12
data type
definition • 2–1
representation in memory (G_ and D_ floating) • C–3, C–4, C–5 to C–6
storage requirement • 2–2 to 2–3
D_floating and G_floating
implementations • 2–4
REAL*8 float intrinsic function • D–38
REAL*8 product of REAL*4's function • D–39
Real editing (F,E,D,G) • 8–17 to 8–25
complex data editing • 8–25
relationship to DECODE statement • A–1
REAL intrinsic function • D–37, D–39
Real Part of Complex function • D–39
REC
DELETE statement keyword • 9–36
specifier in I/O statements • 7–6
RECL
INQUIRE statement specifier • 9–32
OPEN statement keyword • 9–4, 9–18
Record and field references • 2–36 to 2–37
examples • 2–38 to 2–39
Record names
statements that can use • 4–30
Records
allowable operations on aggregate fields • 2–37
arrangement in memory • 2–33 to 2–36
defining values of • 3–1
freeing locked records • 9–38
general description • 2–31 to 2–32
I/O records
deleting records from a file (DELETE) • 9–36 to 9–37
RECORDTYPE keyword (INQUIRE statement) • 9–32
RECORDTYPE keyword (OPEN statement) • 9–20

Records
I/O records (cont'd.)
sizes (OPEN statement keywords) • 9–18
RECORDSIZE
OPEN statement keyword • 9–4, 9–20
Record size (RECL)
default values • 9–19
limits • 9–19
Record specifier
control list parameter
in I/O statements • 7–6
RECORD statement
general description • 4–29 to 4–30
RECORDTYPE
INQUIRE statement specifier • 9–32
OPEN statement keyword • 9–4, 9–20 to 9–21
%REF built-in function • 6–10
References, function
See function references
References, generic or specific
See function references
Relational expressions • 2–48 to 2–49
Relational operators • 2–48
avoiding use as field names • 2–38
in expressions • D–1
Relative organization files
defining size and structure
DEFINE FILE statement • A–3 to A–4
deleting records from
DELETE statement • 9–36 to 9–37
freeing locked records • 9–38
Remainder intrinsic function • D–41
Repeat count
in FORMAT statements • 8–8
Return argument, alternate • 6–9
RETURN statement
general description • 5–25 to 5–27
in parallel DO-loop • 10–4
return args in CALL • 5–25, 5–27
use with FUNCTION statement • 6–15
use with SUBROUTINE statement • 6–18, 6–20
REWIND statement
See also BACKSPACE statement
general description • 9–34
REWRITE statements • 7–45 to 7–47
Run-time formats • 8–41 to 8–42
Run-time library routines
in parallel DO-loop • 10–4

S

S
edit descriptor • 8–12
SAVE
file disposition • 9–23
SAVE statement
general description • 4–30 to 4–31
use of for private entities • 10–7
using unsubscripted array names • 2–29
Scalar field reference
examples • 2–38 to 2–39
format of • 2–36 to 2–37
Scalar memory reference • 2–39 to 2–41
Scalar reference • 2–39 to 2–41
Scale factor editing (P) • 8–25 to 8–27
SECNDS function subprogram • D–48
Segmented records
RECORDTYPE keyword (OPEN statement) • 9–4, 9–20 to 9–21
Separators
external field separators • 8–40 to 8–41
format specification
separators • 8–39 to 8–40
Sequence number field • 1–10, 1–15
SEQUENTIAL
INQUIRE statement specifier • 9–33
OPEN statement keyword value • 9–4
Sequential I/O statements
READ statements • 7–15 to 7–25
WRITE statements • 7–33 to 7–39
Sequential organization files
freeing locked records • 9–38
repositioning
REWIND statement • 9–34
repositioning with BACKSPACE
statement • 9–35
writing end-of-file records
ENDFILE statement • 9–35 to 9–36
'SEQUENTIAL'
OPEN statement keyword value • 9–8
SHARED
OPEN statement keyword • 9–4, 9–21

SHARED directive • 10–7
SHARED_ALL directive • 10–8
Sign control editing • 8–11
SIGN intrinsic function • D–41
Simple list elements
I/O list parameter
in I/O statements • 7–12
SIND intrinsic function • D–33
Sine (degree) intrinsic function • D–33
Sine intrinsic function • D–33
SINH intrinsic function • D–35
SIN intrinsic function • D–33
SIZEOF intrinsic function • D–44
Slash (/)
division operator • 2–42 to 2–44, 2–51
record terminators
in FORMAT statements • 8–4
Source code
See also source program
allowable characters • 1–9
comments in • 1–7
debugging statements in • 1–14
description of fields • 1–9 to 1–15
format using fixed-format • 1–10 to 1–11
format using tab-format • 1–11 to 1–13
Source listing
of CDD records • 1–16
of included files • 1–17
Source program
See also source code
definition of a program unit • 1–2
D in column 1 • 1–14
statement order • 1–3 to 1–5
symbolic names in • 1–5 to 1–7
SP
edit descriptor • 8–11
Space character • 1–9
effect of FORMAT descriptors • 8–7, 8–10
in statement label fields • 1–13
Special characters • 1–8
Specification statements • 4–1 to 4–42
SQRT intrinsic function • D–33
Square root intrinsic function • D–33
SS
edit descriptor • 8–11
/STANDARD qualifier • 4–31
Standards
See ANSI Standard, FORTRAN-66, FORTRAN–77, ISO Standard
Statement field • 1–10, 1–14
Statement functions • 6–12 to 6–14
Statement label
rules governing use of • 1–13
Statement label field • 1–10, 1–13
Statement label references
FORMAT and GOTO statements • 3–7
symbolic • 3–7 to 3–8
Statement labels
assigning symbols to • 3–7 to 3–8
rules governing use • 1–3
Statement order
following OPTIONS statement • 1–19
requirements of • 1–3t
Statements
See FORTRAN statements
STATUS
CLOSE statement keyword • 9–23
OPEN statement keyword • 9–4, 9–21
STOP statement
general description • 5–27
in parallel DO-loop • 10–4
Storage allocation, file
OPEN statement keywords
EXTENDSIZE • 9–4, 9–12
INITIALSIZE • 9–4, 9–14
Storage requirements
of data types • 2–3
Storage units
character • 2–2
numeric • 2–2
Stream records
RECORDTYPE keyword (OPEN statement) • 9–4, 9–20 to 9–21
Structure declaration block
general description • 2–32 to 2–33
Structure declaration blocks
components of • 4–32
data type declaration rules • 4–34
field declarations within • 4–34 to 4–38
general description • 4–31 to 4–41
use of %FILL • 2–33, 4–34
Structure declarations • 4–33
STRUCTURE statement
general description • 4–33 to 4–34

Subexpressions
in logical expressions • 2–51
SUBMIT
file disposition • 9–23
Subprogram arguments
aggregate field references used as • 2–37
associating arrays with • 2–25
associating variables with • 2–22
bit function arguments • D–50 to D–52
external procedure names used as • 4–21 to 4–22
general description • 6–2 to 6–11
adjustable arrays • 6–3 to 6–5
alternate return arguments • 6–9
assumed-size arrays • 6–6
character arrays • 6–7
defaults for arguments passing • 6–9 to 6–11
Hollerith and character constants • 6–8
overview • 6–2
passed-length character arguments • 6–7
intrinsic function names used as • 4–23
use of built-in functions
argument list functions (%VAL, %REF, %DESCR) • 6–9 to 6–11
%LOC function • 6–11
Subprograms
bit functions
general discussion about • D–50 to D–52
CHARACTER FUNCTION statement • 6–15
definition of • 1–2
effect of END statement • 5–12
ENTRY statement • 6–21 to 6–24
function references • 6–16 to 6–18
functions, built-in
argument list functions (%VAL, %REF, %DESCR) • 6–9 to 6–11
%LOC function • 6–11
FUNCTION statement • 6–15 to 6–18
invoking with CALL • 5–2
passed-length character arguments used in • 2–23
SUBROUTINE statement • 6–18 to 6–21
system-supplied FORTRAN intrinsic functions
algorithms used in • D–32
character comparison functions • 6–30 to 6–32
Subprograms
system-supplied FORTRAN intrinsic functions (cont'd.)
complete list of • D–32 to D–45
description of types • 6–2
duplicating external procedure names • 4–21
lexical comparison functions • 6–32 to 6–33
references, generic • 6–26 to 6–27, 6–28 to 6–30
references, specific • 6–25, 6–28 to 6–30
system-supplied intrinsic functions
names used as arguments • 4–23
system-supplied subroutines and functions
list and descriptions of • D–45 to D–53
use of RETURN statement • 5–25 to 5–27
user-written functions
function subprograms • 6–15 to 6–18
subroutine subprograms • 6–18 to 6–21
user-written subprograms
general description • 6–11 to 6–12
statement functions • 6–12 to 6–14
Subroutine arguments
See subprogram arguments
SUBROUTINE statement • 6–18 to 6–21
see also subprograms
position after OPTIONS statement • 1–19
using unsubscripted array names • 2–29
Subscripts, array • 2–27
Substrings
making equivalent • 4–17 to 4–20
Substrings, character
addressing in variables, arrays, and array elements • 2–30 to 2–31
definition • 2–4
Substructure
example of • 2–35
Substructure declarations
general description • 2–32, 4–32, 4–34, 4–38
SUBTITLE directive • 10–13
Subtraction operator (–) • 2–42 to 2–44
Symbolic names
assigning to constants with PARAMETER statement • 4–26 to 4–28
assigning to main program unit • 4–28 to 4–29

Symbolic names (cont'd.)
default data types • 4–22
external procedure names as subprogram arguments • 4–21 to 4–22
intrinsic function names used as subprogram arguments • 4–23
of arrays • 2–25
of constants • 4–26
rules, conventions, and use • 1–5 to 1–7
use with variables • 2–22
Symbolic statement labels
establishing • 3–7 to 3–8
in formatted I/O statements • 3–7 to 3–8
in GOTO statements • 3–7 to 3–8
Symbols
in parallel processing
specifying private default shareability • 10–3, 10–7
specifying private shareability • 10–3, 10–6
modifying
shareability in parallel DO-loop • 10–2 to 10–7
System services
calling from parallel DO-loop • 10–4
System time
function subprogram for calculating SECNDS • D–48
subroutine for calculating TIME • D–49 to D–50

T

Tab format • 1–11 to 1–13
Tags
of compiler directives • 10–1
TAND intrinsic function • D–34
Tangent (degree) intrinsic function • D–34
Tangent intrinsic function • D–34
TANH intrinsic function • D–35
TAN intrinsic function • D–34
T edit descriptor • 8–34, 8–35
Text file libraries
accessing • 1–17 to 1–18
Time, system
See System time • D–48
TIME subroutine • D–49 to D–50
TITLE directive • 10–13
TL edit descriptor • 8–34, 8–36
Transfer, control
See Control transfer
Transfer-of-control specifier
control list parameter
in I/O statements • 7–10
Transfer of Sign intrinsic function • D–41
TR edit descriptor • 8–34, 8–36
Truncation intrinsic function • D–36
TYPE
OPEN statement keyword • 9–4, 9–22
Type declaration statement
See Data type declaration statement
to establish variables • 2–22, 2–23
TYPE statement • 7–48 to 7–49

U

Unary operators
definition of • 2–43
Unary plus and minus operators (+ and –) • 2–42 to 2–44, 2–51
Unconditional GO TO statement • 5–13
Undeclared symbolic names
default data types • 4–22
UNFORMATTED
INQUIRE statement specifier • 9–33
Unformatted I/O statements
READ statements
direct access • 7–26, 7–27
indexed • 7–28, 7–30
sequential • 7–15, 7–25
REWRITE statements • 7–45, 7–46
use of aggregate field references • 2–37
WRITE statements
direct access • 7–39, 7–40
indexed • 7–41, 7–43
sequential • 7–33, 7–39
Union declarations
contrasted with EQUIVALENCE • 4–40
definition • 4–32
general description • 4–38 to 4–41
UNION statement • 4–39
UNIT
BACKSPACE statement keyword • 9–35
CLOSE statement keyword • 9–23

UNIT (cont'd.)
DELETE statement keyword • 9–36
ENDFILE statement keyword • 9–35
INQUIRE statement keyword • 9–25
OPEN statement keyword • 9–4, 9–22
REWIND statement keyword • 9–34
specifier in I/O statements • 7–3
UNLOCK statement keyword • 9–38
UNLOCK statement • 9–38
Unnamed fields
in a structure • 2–33
Unsubscripted array names
usage restrictions • 2–29
use of • 2–29
Uppercase characters • 1–8
effect on compiler • 1–9
in character and Hollerith constants • 1–9
User-defined functions
references to in dimension bounds expressions • 2–26
USEROPEN
OPEN statement keyword • 9–4, 9–23

V

%VAL built-in function • 6–10
Variable-length records
RECORDTYPE keyword (OPEN statement) • 9–4, 9–20 to 9–21
Variables
as scalar references • 2–39
association of two or more • 2–22
character substrings • 2–30 to 2–31
data types of • 2–1
data typing by implication • 2–24
data typing by specification • 2–23
default shareability in parallel DO-loop • 10–2 to 10–7
defining in memory • 2–22
defining values of • 3–1
definition • 2–4, 2–22
establishing with subprogram references • 2–22
initializing with DATA statements • 4–5 to 4–8
in structure declarations • 2–32
Variant record capability • 2–33
VAX FORTRAN
extensions to ANSI Standard • 1–1
VIRTUAL statement
DIMENSION statement compared to • 4–13
VOLATILE statement
general description • 4–41

W

/WARNINGS qualifier • 4–23
WRITE statements • 7–33 to 7–44
direct access WRITE • 7–39 to 7–40
formatted • 7–39, 7–40
unformatted • 7–39, 7–40
indexed WRITE • 7–41 to 7–43
formatted • 7–41, 7–42
unformatted • 7–41, 7–43
internal WRITE • 7–43 to 7–44
formatted • 7–43, 7–44
list-directed • 7–43, 7–44
relationship to ENCODE statement • A–1, A–2
sequential WRITE • 7–33 to 7–39
formatted • 7–33, 7–34 to 7–35
list-directed • 7–33, 7–35 to 7–37
namelist-directed • 7–33, 7–37 to 7–38
unformatted • 7–33, 7–39

X

X edit descriptor • 8–34, 8–34 to 8–35
.XOR.
See logical operators

Z

Zero-Extended Functions • D–37
ZEXT intrinsic function • D–37
Z field descriptor • 8–16

Reader's Comments

VAX FORTRAN
Language Reference Manual
AA–D034E–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

-- 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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here ----------

Cut Along Dotted Line

Reader's Comments

VAX FORTRAN
Language Reference Manual
AA–D034E–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

-- 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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here ----------

Cut Along Dotted Line

Reader's Comments

VAX FORTRAN
Language Reference Manual
AA–D034E–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

Do Not Tear - Fold Here

Cut Along Dotted Line

Reader's Comments

VAX FORTRAN
Language Reference Manual
AA–D034E–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

-- 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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here ----------

Cut Along Dotted Line