

VAX C

digital

Guide to VAX C

Order Number: AA-L370D-TE

Guide to VAX C

Order Number: AA-L370D-TE

February 1989

This document describes VAX C constructs in context with both the history of the C programming language and that of the VMS environment. It contains information on VAX C program development in the VMS environment, the VAX C programming language, and cross-system portability concerns.

Revision/Update Information: This revised manual supersedes the *Guide to VAX C* (Order No. AA-L370C-TE).

Operating System and Version: VMS Version 5.0 or higher

Software Version: VAX C Version 3.0

**digital equipment corporation
maynard, massachusetts**

First Printing, May 1982
Revised, April 1985
Revised, March 1987
Revised, January 1989

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.

© Digital Equipment Corporation 1982, 1985, 1987, 1989.

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

The postpaid Reader's Comments forms at the end of this document request your 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-10	PDT	
DECUS	RSTS	
DECwriter	RSX	

ZK4565

Contents

Preface	xxi
New and Changed Features	xxv

Developing VAX C Programs on VMS Systems

Chapter 1	Developing VAX C Programs at the DCL Command Level	
1.1	DCL Commands for Program Development	1-1
1.2	Creating a VAX C Program	1-4
1.2.1	Using VAXTPU	1-4
1.2.1.1	The EVE Interface	1-4
1.3	Compiling a VAX C Program	1-5
1.3.1	The CC Command	1-5
1.3.2	The CC Command Qualifiers	1-7
1.3.2.1	Using the /DEFINE and /UNDEFINE Qualifiers ...	1-18
1.3.3	Compiler Error Messages	1-20
1.4	Linking a VAX C Program	1-22
1.4.1	The LINK Command	1-22
1.4.2	LINK Command Qualifiers	1-23
1.4.3	Linker Input Files	1-24
1.4.4	Linker Output Files	1-25

1.4.5	Linking Against Object Module Libraries and Shareable Images	1-26
1.4.5.1	Object Module Libraries	1-26
1.4.5.2	Linking Against the RTL Object Libraries	1-27
1.4.5.3	Linking Against the RTL Shareable Images	1-29
1.4.6	Linker Error Messages	1-30
1.5	Running a VAX C Program	1-31

Chapter 2 Using the VMS Debugger

2.1	Overview	2-1
2.2	Features of the Debugger	2-3
2.3	Getting Started with the Debugger	2-4
2.3.1	Compiling and Linking a Program to Prepare for Debugging . .	2-4
2.3.2	Starting and Terminating a Debugging Session	2-5
2.3.3	Aborting Program Execution or Debugger Commands	2-6
2.3.4	Entering Debugger Commands	2-7
2.3.5	Viewing Your Source Code	2-10
2.3.5.1	Noscreen Mode	2-10
2.3.5.2	Screen Mode	2-11
2.3.6	Controlling and Monitoring Program Execution	2-12
2.3.6.1	Starting and Resuming Program Execution	2-12
2.3.6.2	Determining Where Execution Is Suspended—SHOW CALLS	2-14
2.3.6.3	Suspending Program Execution	2-15
2.3.6.4	Tracing Program Execution	2-17
2.3.6.5	Monitoring Changes in Variables	2-18
2.3.7	Examining and Manipulating Data	2-19
2.3.7.1	Displaying the Values of Variables	2-20
2.3.7.2	Changing the Values of Variables	2-21
2.3.7.3	Evaluating Expressions	2-21
2.4	Notes on Debugger Support for VAX C	2-23
2.4.1	Debugger Command-Line Options	2-23
2.4.2	Accessing Scalar Variables	2-23
2.4.3	Accessing Arrays	2-25
2.4.4	Accessing Character Strings	2-27
2.4.5	Accessing Structures and Unions	2-28
2.5	Controlling Symbol References	2-34
2.5.1	Module Setting	2-34

2.5.2	Resolving Multiply Defined Symbols	2-35
2.6	Sample Debugging Session	2-36

Chapter 3 VAX C Support for Parallel Processing

3.1	Overview of Parallel Processing	3-1
3.2	Preparing Programs for Parallel Processing	3-6
3.3	Conditions That Inhibit Parallel Processing	3-9
3.4	Data-Dependency Analysis	3-11
3.4.1	Array Variable References	3-11
3.4.2	Function Calls	3-13
3.4.2.1	math.h Function Calls	3-14
3.4.3	Pointer Variable References	3-15
3.4.4	Scalar Variable References	3-16
3.5	Rewriting Code to Resolve Dependencies	3-17
3.5.1	Loop Alignment	3-18
3.5.2	Code Replication	3-20
3.5.3	Loop Distribution	3-21
3.6	Storage Classes and Parallel Processing	3-22
3.7	Decomposition Pragas	3-23
3.7.1	The ignore_dependency Decomposition Pragma	3-25
3.7.2	The safe_call Decomposition Pragma	3-26
3.7.3	The sequential_loop Decomposition Pragma	3-28
3.8	Memory-Management Functions	3-29
3.9	Tuning Issues Related to Parallel Processing	3-30
3.9.1	Customizing the Parallel-Processing Run-Time Environment ..	3-30
3.9.1.1	Controlling the Number of Processes (FOR\$PROCESSES)	3-31
3.9.1.2	Controlling Internal Spin Waits (FOR\$SPIN_WAIT)	3-32
3.9.1.3	Controlling the State of a Process (FOR\$STALL_WAIT)	3-33

3.9.2	System Parameters Set with the SYSGEN Utility	3-33
3.9.2.1	Global Section Descriptor Count (GBLSECTIONS)	3-34
3.9.2.2	Global Page Table Entry Count (GBLPAGES)	3-35
3.9.2.3	Global Page File Limit (GBLPAGFIL)	3-35
3.9.3	User Parameters Set with the AUTHORIZE Utility	3-36
3.9.4	Other Tuning Considerations	3-37

VAX C Programming Concepts

Chapter 4 VAX C Tutorial

4.1	C Programming Language Overview	4-1
4.2	VAX C Programming Language Overview	4-3
4.3	Writing a Program	4-4
4.4	Producing Input/Output (I/O)	4-6
4.5	Conditional Execution of Code	4-10
4.5.1	The if Statement	4-10
4.5.2	The switch Statement	4-12
4.5.3	Loops	4-14
4.6	Values, Addresses, and Pointers	4-17
4.7	Aggregates	4-21
4.7.1	Arrays and Character Strings	4-21
4.7.2	Structures and Unions	4-22

Chapter 5 Program Structure

5.1	Function Definitions	5-1
5.1.1	Main Function and Function Identifiers	5-3
5.1.2	Parameter List Declarations	5-4
5.1.3	Function Return Data Types	5-5
5.1.4	Variable-Length Parameter Lists	5-6
5.2	Function Declarations	5-7

5.3	Function Prototypes	5-9
5.3.1	Using Function Prototypes	5-11
5.4	Using Parameters and Arguments	5-12
5.4.1	Function and Array Identifiers as Arguments	5-13
5.4.2	Passing Arguments to the main Function	5-15
5.5	Identifiers	5-17
5.6	Language Keywords	5-18
5.7	Blocks	5-21
5.8	Comments	5-22
5.9	LINT-Like Functionality	5-22

Chapter 6 **Statements**

6.1	Control Flow Statements	6-1
6.1.1	The null Statement	6-1
6.1.2	The goto Statement	6-2
6.1.3	The label Statement	6-2
6.2	Expressions and Blocks as Statements	6-3
6.2.1	The expression Statement	6-3
6.2.2	The compound Statement	6-3
6.3	Conditional Statements	6-4
6.3.1	The if Statement	6-4
6.3.2	The switch Statement	6-5
6.3.2.1	Declarations Within a switch Statement	6-7
6.4	Looping Statements	6-7
6.4.1	The for Statement	6-8
6.4.2	The while Statement	6-9
6.4.3	The do Statement	6-9
6.5	Interrupting Statements	6-10
6.5.1	The break Statement	6-10
6.5.2	The continue Statement	6-10
6.5.3	The return Statement	6-11

Chapter 7 Expressions and Operators

7.1	lvalues and rvalues	7-2
7.2	Primary Expressions and Operators	7-3
7.2.1	Parenthetical Expressions	7-3
7.2.2	Function Calls	7-3
7.2.3	Array References	7-5
7.2.4	Structure and Union References	7-6
7.3	Overview of the VAX C Operators	7-6
7.4	Unary Expressions and Operators	7-10
7.4.1	Negating Arithmetic and Logical Expressions	7-10
7.4.2	Incrementing and Decrementing Variables	7-10
7.4.3	Computing Addresses and Dereferencing Pointers	7-11
7.4.4	Calculating a One's Complement	7-12
7.4.5	Forcing Conversions to a Specific Type	7-13
7.4.6	Calculating Sizes of Variables and Data Types	7-14
7.5	Binary Expressions and Operators	7-14
7.5.1	Additive Operators	7-15
7.5.2	Multiplication Operators	7-15
7.5.3	Equality Operators	7-16
7.5.4	Relational Operators	7-16
7.5.5	Bitwise Operators	7-17
7.5.6	Logical Operators	7-17
7.5.7	Shift Operators	7-19
7.6	Conditional Operator	7-19
7.7	Assignment Expressions and Operators	7-20
7.8	Comma Expression and Operator	7-22
7.9	Data-Type Conversions	7-22
7.9.1	Converting Operands	7-23
7.9.2	Converting Function Arguments	7-24

Chapter 8 Data Types and Declarations

8.1	Constants	8-1
8.2	Variables	8-2
8.2.1	Classification of Variables	8-2
8.2.1.1	Data-Type Keywords	8-3
8.2.1.2	Format of a Variable Declaration	8-3
8.3	Integers (int, long, short, char, and unsigned)	8-4
8.3.1	Integer Constants	8-5
8.3.2	Character Constants	8-6
8.3.3	Escape Sequences	8-7
8.4	Floating-Point Numbers (float and double)	8-9
8.4.1	Floating-Point Constants	8-10
8.5	Pointers	8-11
8.5.1	void Pointers	8-13
8.6	Enumerated Types (enum)	8-13
8.7	Arrays ([])	8-15
8.7.1	Initializing Arrays	8-18
8.8	Character-String Variables (char * and char [])	8-19
8.8.1	Character-String Constants	8-20
8.9	Structures and Unions (struct and union)	8-20
8.9.1	Declaring a Structure or Union	8-22
8.9.2	Referencing Members of Structures or Unions	8-24
8.9.3	Initializing Structures and Unions	8-26
8.9.4	Variant Structures and Unions	8-28
8.9.5	Bit Fields	8-30
8.10	The void Keyword	8-32
8.11	The typedef Keyword	8-32
8.12	Interpreting Declarations	8-33

Chapter 9 Storage Classes and Allocation

9.1	The Scope of an Identifier	9-1
9.1.1	The Compilation and Linking Process	9-2
9.1.2	Position of the Declaration	9-2
9.1.3	Lexical Scope and Link-Time Scope	9-4
9.1.4	Program Example	9-6
9.2	Storage Allocation	9-8
9.3	Internal Storage Classes	9-9
9.3.1	The auto Specifier	9-10
9.3.2	The register Specifier	9-11
9.4	Static Storage Class	9-12
9.5	External Storage Class	9-13
9.6	Global Storage Classes	9-15
9.6.1	The globaldef and globalref Specifiers	9-15
9.6.1.1	Comparing the Global and the External Storage Classes	9-17
9.6.2	The globalvalue Specifier	9-19
9.6.3	Global Enumerated Types	9-20
9.7	Data-Type Modifiers	9-21
9.7.1	The const Modifier	9-21
9.7.2	The volatile Modifier	9-23
9.8	Storage-Class Modifiers	9-23
9.8.1	The noshare Modifier	9-24
9.8.2	The readonly Modifier	9-25
9.8.3	The _align Modifier	9-25

Chapter 10 Preprocessor Directives

10.1	Macro Definitions (#define and #undef)	10-2
10.1.1	Constant Identifiers	10-4
10.1.2	Canceling Definitions (#undef)	10-4
10.1.3	Macro Parameters	10-4
10.1.4	Listing Substituted Lines	10-8
10.2	Common Data Dictionary Extraction (#dictionary)	10-8

10.2.1	Using the #dictionary Directive	10-9
10.2.2	Support for CDD Data Types	10-11
10.3	Conditional Compilation (#if, #ifdef, #ifndef, #else, #elif, and #endif)	10-13
10.3.1	The defined Operator	10-15
10.4	File Inclusion (#include)	10-16
10.4.1	Inclusion Using Angle Brackets	10-17
10.4.2	Inclusion Using Quotation Marks (" ")	10-18
10.4.3	Inclusion of Text Modules	10-19
10.4.4	Macro Substitution in #include Directives	10-20
10.5	Specifying Line Numbers (#line and #)	10-21
10.6	Specifying the Module Name and Identification (#module)	10-21
10.7	Implementation-Specific Preprocessor Directive (#pragma)	10-22
10.7.1	#pragma [no]builtins Directive	10-23
10.7.2	#pragma ignore_dependency Directive	10-23
10.7.3	#pragma [no]inline	10-24
10.7.3.1	Restrictions on Inline Expansion	10-25
10.7.4	#pragma [no]member_alignment	10-25
10.7.5	#pragma safe_call Directive	10-26
10.7.6	#pragma sequential_loop Directive	10-27
10.7.7	#pragma [no]standard Directive	10-28

Chapter 11 Predefined Macros and Built-In Functions

11.1	Predefined Macros	11-1
11.1.1	CC\$gfloat (G_Floating Identification Macro)	11-1
11.1.2	CC\$parallel (Parallel-Processing Identification Macro)	11-2
11.1.3	The __DATE__ Macro	11-3
11.1.4	The __FILE__ Macro	11-3
11.1.5	The __LINE__ Macro	11-3
11.1.6	The __TIME__ Macro	11-3
11.1.7	vax, vms, vaxc, and vax11c (System-Identification Macros)	11-4
11.2	Built-In Functions	11-4
11.2.1	Add Aligned Word Interlocked (_ADAWI)	11-5
11.2.2	Branch on Bit Clear-Clear Interlocked (_BBCCI)	11-6
11.2.3	Branch on Bit Set-Set Interlocked (_BBSSI)	11-6
11.2.4	Find First Clear Bit (_FFC)	11-7

11.2.5	Find First Set Bit (<code>_FFS</code>)	11-8
11.2.6	Halt (<code>_HALT</code>)	11-8
11.2.7	Insert Entry into Queue at Head Interlocked (<code>_INSQHI</code>)	11-9
11.2.8	Insert Entry into Queue at Tail Interlocked (<code>_INSQTI</code>)	11-9
11.2.9	Insert Entry in Queue (<code>_INSQUE</code>)	11-10
11.2.10	Load Process Context (<code>_LDPCTX</code>)	11-10
11.2.11	Locate Character (<code>_LOCC</code>)	11-10
11.2.12	Move from Processor Register (<code>_MFPR</code>)	11-11
11.2.13	Move Character 3 Operand (<code>_MOVC3</code>)	11-11
11.2.14	Move Character 5 Operand (<code>_MOVC5</code>)	11-12
11.2.15	Move from Processor Status Longword (<code>_MOVPSL</code>)	11-13
11.2.16	Move to Processor Register (<code>_MTPR</code>)	11-14
11.2.17	Probe Read Accessibility (<code>_PROBER</code>)	11-14
11.2.18	Probe Write Accessibility (<code>_PROBEW</code>)	11-15
11.2.19	Read General-Purpose Register (<code>_READ_GPR</code>)	11-15
11.2.20	Remove Entry from Queue at Head Interlocked (<code>_REMQHI</code>)	11-16
11.2.21	Remove Entry from Queue at Tail Interlocked (<code>_REMQTI</code>)	11-16
11.2.22	Remove Entry from Queue (<code>_REMQUE</code>)	11-17
11.2.23	Scan Characters (<code>_SCANC</code>)	11-17
11.2.24	Simple Read (<code>_SIMPLE_READ</code>)	11-18
11.2.25	Simple Write (<code>_SIMPLE_WRITE</code>)	11-19
11.2.26	Skip Character (<code>_SKPC</code>)	11-19
11.2.27	Span Characters (<code>_SPANC</code>)	11-20
11.2.28	Save Process Context (<code>_SVPCTX</code>)	11-21
11.2.29	Write General-Purpose Register (<code>_WRITE_GPR</code>)	11-21

Using VAX C Features on VMS Systems

Chapter 12 Using VAX Record Management Services

12.1	RMS File Organization	12-2
12.1.1	Sequential File Organization	12-2
12.1.2	Relative File Organization	12-3
12.1.3	Indexed File Organization	12-3
12.2	Record Access Modes	12-4
12.3	RMS Record Formats	12-5
12.4	RMS Functions	12-5

12.5	Writing VAX C Programs Using RMS	12-7
12.5.1	Initializing File Access Blocks	12-9
12.5.2	Initializing Record Access Blocks	12-10
12.5.3	Initializing Extended Attribute Blocks	12-11
12.5.4	Initializing Name Blocks	12-12
12.6	RMS Example Program	12-13

Chapter 13 Using VAX C in the Common Language Environment

13.1	The VAX Procedure Calling and Condition Handling Standard	13-2
13.1.1	Register and Stack Usage	13-3
13.1.2	Return of the Function Value	13-5
13.1.3	The Argument List	13-5
13.2	Specifying Parameter-Passing Mechanisms	13-6
13.2.1	Passing Arguments by Immediate Value	13-8
13.2.2	Passing Arguments by Reference	13-11
13.2.3	Passing Arguments by Descriptor	13-14
13.2.4	VAX C Default Parameter-Passing Mechanisms	13-19
13.3	Interlanguage Calling	13-19
13.3.1	Calling VAX FORTRAN	13-20
13.3.2	Calling VAX MACRO	13-25
13.3.3	Calling VAX BASIC	13-29
13.3.4	Calling VAX Pascal	13-32
13.4	Sharing Global Data	13-37
13.4.1	Sharing Program Sections with FORTRAN Common Blocks	13-37
13.4.2	Sharing Program Sections with PL/I Externals	13-39
13.4.3	Sharing Program Sections with MACRO Programs	13-41
13.5	VMS Run-Time Library Routines	13-42
13.6	VMS System Services Routines	13-43
13.7	Calling Routines	13-44
13.7.1	Determining the Type of Call	13-44
13.7.2	Declaring an External Routine and Its Arguments	13-45
13.7.3	Calling the External Routine	13-45
13.7.4	System Routine Arguments	13-45
13.7.5	Symbol Definitions	13-49
13.7.6	Condition Values	13-50

13.7.7	Checking System Service Return Values	13-50
13.8	Variable-Length Argument Lists in System Services	13-52
13.9	Return Status Values	13-54
13.9.1	Format of Return Status Values	13-54
13.9.2	Manipulating Return Status Values	13-56
13.9.3	Testing for Success or Failure	13-58
13.9.4	Testing for Specific Return Status Values	13-59
13.10	Examples of Calling System Routines	13-61

Chapter 14 VAX C Implementation Notes

14.1	Program Sections	14-1
14.1.1	Attributes of Program Sections (Psects)	14-1
14.1.2	Program Sections Created by VAX C	14-2

Appendix A VAX C Definition Modules

Appendix B VAX C Compiler Messages

Appendix C Optional Programming/ Productivity Tools

C.1	Using VAX LSE with VAX C	C-1
C.1.1	Entering Source Code Using Tokens and Placeholders	C-2
C.1.2	Compiling Source Code	C-4
C.1.2.1	Pragma Insertions and Decomposition	C-5
C.1.3	Examples	C-6
C.1.3.1	Preprocessor Lines	C-7
C.1.3.2	External Definition	C-7
C.1.3.3	Function Definition	C-8
C.1.3.4	Block Declaration	C-11
C.1.3.5	Statements and Expressions	C-17
C.2	Using the VAX Source Code Analyzer	C-20
C.2.1	Multimodular Development	C-21

C.2.2	Setting Up an SCA Environment	C-23
	C.2.2.1 Creating an SCA Library	C-23
	C.2.2.2 Generating the Data Analysis Files	C-24
	C.2.2.3 Selecting an SCA Library	C-24
	C.2.2.4 Loading Data Analysis Files into a Local Library	C-24
C.2.3	Using SCA for Cross-Referencing	C-25

Appendix D Language Summary

D.1	The CC Command	D-1
D.2	The LINK Command	D-3
D.3	Data-Type Keywords	D-4
D.4	Precedence of Operators	D-5
D.5	Statements	D-6
D.6	Conversion Rules	D-7
D.7	Escape Sequences	D-8
D.8	Preprocessor Directives	D-8
D.9	Record Management Services (RMS)	D-9

Appendix E Working with the Multiprocess Debugging Configuration

E.1	Getting Started	E-1
	E.1.1 Establishing a Multiprocess Debugging Configuration	E-2
	E.1.2 Invoking the Debugger	E-2
	E.1.3 The Visible Process and Process-Specific Commands	E-3
	E.1.4 Obtaining Information About Processes	E-3
	E.1.5 Bringing a Spawned Process Under Debugger Control	E-5
	E.1.6 Broadcasting Commands to Selected Processes	E-6
	E.1.7 Controlling Execution	E-7
	E.1.7.1 Controlling Execution with SET MODE NOINTERRUPT	E-8
	E.1.7.2 Putting Selected Processes on Hold	E-8
E.1.8	Changing the Visible Process	E-9

E.1.9	Dynamic Process Setting	E-10
E.1.10	Monitoring the Termination of Images	E-11
E.1.11	Terminating the Debugging Session	E-11
E.1.12	Releasing Selected Processes from Debugger Control	E-11
E.1.13	Aborting Debugger Commands and Interrupting Program Execution	E-12
E.2	Supplemental Information	E-13
E.2.1	Specifying Processes in Debugger Commands	E-13
E.2.2	Monitoring Process Activation and Termination	E-15
E.2.3	Interrupting the Execution of an Image to Connect It to the Debugger	E-15
E.2.3.1	Using the CTRL/Y-DEBUG Sequence to Invoke the Debugger	E-16
E.2.3.2	Using the CONNECT Command to Interrupt an Image	E-17
E.2.4	Screen Mode Features for Multiprocess Debugging	E-17
E.2.5	Setting Watchpoints in Global Sections	E-19
E.2.6	Compatibility of Multiprocess Commands with the Default Configuration	E-20
E.3	Sample Multiprocess Debugging Session	E-21
E.4	Considerations for Multiprocess Debugging	E-25
E.4.1	User Quotas	E-25
E.4.2	System Resources	E-26

VAX C Glossary

Index

Examples

1-1	Symbol Cross-References in a Compiler Listing	1-9
2-1	Debugging Sample Program SCALARS.C	2-24
2-2	Debugging Sample Program ARRAY.C	2-26
2-3	Debugging Sample Program STRING.C	2-27
2-4	Debugging Sample Program STRUCT.C	2-30
2-5	Debugging Sample Program ARSTRUCT.C	2-32
2-6	Debugging Sample Program POWER.C	2-37

2-7	A Sample Debugging Session	2-37
3-1	Using the #pragma ignore_dependency Directive	3-25
3-2	Using the #pragma ignore_dependency Directive	3-26
3-3	Using the #pragma safe_call Directive	3-27
3-4	Using the #pragma sequential_loop Directive	3-29
4-1	Simple Addition in VAX C	4-4
4-2	Output of Information	4-8
4-3	Output Using the Newline Character	4-9
4-4	Conditional Execution Using the if Statement	4-11
4-5	Conditional Execution Using the switch Statement	4-12
4-6	Looping Using the do Statement	4-14
4-7	Looping Using the for Statement	4-16
4-8	Character-String Constants and Arrays	4-22
4-9	Single Storage Allocation of Unions	4-24
4-10	Structures	4-25
5-1	Case Conversion Program	5-2
5-2	Declaring Functions	5-7
5-3	Declaring Functions Passed as Arguments	5-14
5-4	Echo Program Using Command-Line Arguments	5-16
5-5	Scope of Variable Declarations in Nested Blocks	5-21
6-1	Using switch to Count Blanks, Tabs, and Newlines	6-6
8-1	Rules for Initialization of Structures	8-27
9-1	Scope and Externally Defined Variables	9-6
9-2	Reinitializing Two auto Variables	9-11
9-3	Using Global Variables	9-16
9-4	Using the globalvalue Specifier	9-20
10-1	Nested Substitution Directives	10-3
12-1	External Data Declarations and Definitions	12-14
12-2	Main Program Section	12-16
12-3	Function Initializing RMS Data Structures	12-18
12-4	Internal Functions	12-20
12-5	Utility Function: Adding Records	12-22
12-6	Utility Function: Deleting Records	12-24
12-7	Utility Function: Typing the File	12-25
12-8	Utility Function: Printing the File	12-27
12-9	Utility Function: Updating the File	12-29
13-1	Passing Floating-Point Arguments by Immediate Value	13-11
13-2	Passing Arguments by Reference	13-13

13-3	Passing Arguments by Descriptor	13-17
13-4	Passing Compile-Time String Descriptors	13-18
13-5	VAX C Function Calling a VAX FORTRAN Subprogram	13-21
13-6	VAX FORTRAN Subprogram Calling a VAX C Function	13-23
13-7	VAX C Function Emulating a VAX FORTRAN CHARACTER*(*) Function	13-24
13-8	VAX MACRO Program Calling a VAX C Function	13-26
13-9	VAX C Program Calling a VAX MACRO Program	13-28
13-10	VAX C Function Calling a VAX BASIC Function	13-30
13-11	VAX BASIC Program Calling a VAX C Function	13-31
13-12	VAX C Function Calling a VAX Pascal Routine	13-32
13-13	VAX Pascal Program Calling a VAX C Function	13-35
13-14	Sharing Data with a FORTRAN Program in Named Program Sections . . .	13-38
13-15	Sharing Data with a FORTRAN Program in a VAX C Structure	13-39
13-16	Sharing Data with a PL/I Program in Named Program Sections	13-40
13-17	Sharing Data with a PL/I Program in a VAX C Structure	13-41
13-18	Sharing Data with a MACRO Program in a VAX C Structure	13-42
13-19	Checking System Service Return Values	13-51
13-20	Using Variable-Length Argument Lists	13-53
13-21	Testing for Success	13-58
13-22	Testing for Specific Return Status Values	13-60
13-23	Passing Arguments to System Services	13-62
13-24	Determining \$QIO Completion	13-63
13-25	Using Time Routines	13-64
E-1	VAX C Program Used for Multiprocess Debugging Session	E-21
E-2	Sample Multiprocess Debugging Session	E-24

Figures

1-1	DCL Commands for Developing Programs	1-2
2-1	Debugger Keypad Key Functions	2-9
3-1	Sequential and Parallel Loop Execution Across Time	3-3
3-2	Program Cycle Using Decomposition	3-8
4-1	rvalues, lvalues, and Assigning Pointers	4-19
4-2	The Indirection Operator in Assignments	4-20
7-1	Boolean Algebra and the Bitwise Operators	7-18
8-1	Alignment of Structure Members	8-31
13-1	The Call Stack	13-4

13-2	Structure of a VAX Argument List	13-5
13-3	Example of a VAX Argument List	13-6
13-4	Passing Arguments by Immediate Value	13-10
13-5	Bit Fields Within a Return Status Value	13-55
13-6	Internal Representation of a Status Value	13-57
C-1	Use of SCA for Multimodular Development	C-22

Tables

1-1	Debugger Compilation Options	1-9
1-2	/MACHINE_CODE Qualifier Options	1-13
1-3	/[NO]OPTIMIZE Qualifier Options	1-14
1-4	/SHOW Qualifier Options	1-16
1-5	/WARNINGS Qualifier Options	1-18
1-6	VMS Linker Default File Types for Input Files	1-25
2-1	Supported Operators	2-22
2-2	Unsupported Operators	2-22
3-1	VAX C Parallel-Processing Support Mechanisms	3-5
3-2	VAX C Decomposition Pragmas	3-23
3-3	Logical Names Used for Run-Time Tuning	3-31
3-4	Sysgen Parameters Requiring Changes for Parallel Processing	3-34
5-1	VAX C Keywords	5-19
5-2	VAX C Features Similar to the LINT Utility	5-23
7-1	VAX C Operators	7-7
7-2	Precedence of VAX C Operators	7-9
8-1	VAX C Data-Type Keywords	8-3
8-2	Size and Range of VAX C Integers	8-5
8-3	VAX C Escape Sequences	8-8
9-1	VAX C Storage Classes and Storage-Class Specifiers	9-4
9-2	Scope and the Storage-Class Specifiers	9-5
9-3	The Variables in Example 9-1 and Their Storage Classes	9-7
9-4	Location, Lifetime, and the Storage-Class Keywords	9-9
9-5	Predefined Alignment Constants	9-26
10-1	Mapping Between CDD and VAX C Data Types	10-12
12-1	Common RMS Run-Time Processing Functions	12-6
12-2	VAX C RMS #include Modules	12-8
12-3	RMS Prototype Data Structures	12-9
13-1	VAX Register Usage	13-3

13-2	Status Values of SYS\$SETEF	13-9
13-3	Status Values of SYS\$READEF	13-12
13-4	Valid Class Codes	13-15
13-5	Atomic Data Types	13-16
13-6	Valid Parameter-Passing Mechanisms in VAX C	13-19
13-7	Default Passing Mechanisms	13-20
13-8	Run-Time Library Facilities	13-43
13-9	System Services	13-44
13-10	VAX C Implementation	13-46
13-11	Possible Severity Values	13-56
13-12	Facility Codes	13-59
14-1	Program Section Attributes	14-2
14-2	Combinations of Storage-Class Specifiers and Modifiers	14-3
14-3	Combination Attributes	14-4
A-1	VAX C Definition Modules	A-1
C-1	LSE Placeholders	C-2
C-2	Commands to Manipulate Tokens and Placeholders	C-3
C-3	LSE Commands to Review and Examine Source Code	C-5
C-4	SCA Commands to Use Within LSE	C-26
D-1	Precedence of Operators	D-5
D-2	Escape Sequences	D-8
D-3	RMS Module Names	D-9
D-4	RMS Templates	D-10
E-1	Debugging States	E-4
E-2	Process Specifications	E-13
E-3	Changed and New Keypad Key Functions	E-19

Preface

This guide combines reference information on the VAX C programming language with information necessary for developing and debugging VAX C programs on the VMS operating system. The guide also includes information about porting C programs to and from VMS and other operating systems, as well as the differences between VAX C and other implementations of the language. For more information about porting programs to and from other operating systems, see the *VAX C Run-Time Library Reference Manual*.

Intended Audience

This guide is intended for experienced programmers who need to learn VAX C, for users who need to know the difference between VAX C and other implementations, or for experienced VAX C users who need to reference information. You should be familiar with one high-level language and should have some familiarity with the DIGITAL Command Language (DCL). If you are not familiar with or need to reference information about DCL, see Chapter 1.

Document Structure

This manual has 14 chapters and 5 appendixes. These chapters are grouped into three parts as follows:

Developing VAX C Programs on VMS Systems

- Chapter 1 explains how to create, compile, link, and run a VAX C program.
- Chapter 2 explains how to use the VMS Debugger.
- Chapter 3 explains how to decompose VAX C loops.

VAX C Programming Concepts

- Chapter 4 presents a brief VAX C tutorial.
- Chapter 5 explains program structure.
- Chapter 6 describes VAX C statements.
- Chapter 7 discusses the types of expressions and the operators used in VAX C.
- Chapter 8 explains data types and declarations.
- Chapter 9 describes storage classes and allocation.
- Chapter 10 explains preprocessor directives.
- Chapter 11 describes the predefined macros and the built-in functions.

Using VAX C Features on VMS Systems

- Chapter 12 explains VAX Record Management Services (RMS).
- Chapter 13 describes VMS System Services and VMS Run-Time Library routines.
- Chapter 14 explains program sections (psects) and VAX C storage classes.

Appendixes

- Appendix A describes VAX C definition modules.
- Appendix B lists VAX C compiler messages.
- Appendix C provides an overview of the VAX Language-Sensitive Editor (LSE) and information on the VAX Source Code Analyzer (SCA).
- Appendix D provides a summary of all VAX C language features.
- Appendix E explains how to debug a program that takes advantage of parallel-processing features.
- The VAX C Glossary provides an alphabetical listing of key terms.

Associated Documents

You may find the following documents useful when programming in VAX C:

- *VAX C Installation Guide*—For system programmers who install the VAX C software.
- *VAX C Run-Time Library Reference Manual*—For programmers who wish to use the VAX C Run-Time Library functions and who need more information about porting programs to and from other operating systems.
- *The C Programming Language*¹ —For those who need a more intensive tutorial than that provided in Chapter 4. This book describes draft-proposed ANSI C. VAX C contains features and enhancements to the C language as described in *The C Programming Language*. Therefore, use the *Guide to VAX C* as the reference book for the full description of VAX C.

Conventions

Convention	Meaning
<code>RETURN</code>	The symbol <code>RETURN</code> represents a single stroke of the RETURN key on a terminal.
<code>CTRL/X</code>	The symbol <code>CTRL/X</code> , where letter X represents a terminal control character, is generated by holding down the CTRL key while pressing the key of the specified terminal character.
<code>\$ RUN CPROG RETURN</code>	In interactive examples, the user's response to a prompt is printed in red; system prompts are printed in black.
<code>float x;</code> <code> .</code> <code> .</code> <code> .</code> <code>x = 5;</code>	A vertical ellipsis indicates that not all of the text of a program or program output is illustrated. Only relevant material is shown in the example.

¹ Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language, Second Edition* (Englewood Cliffs, New Jersey: Prentice Hall, 1988).

Convention	Meaning
option, ...	A horizontal ellipsis indicates that additional parameters, options, or values can be entered. A comma that precedes the ellipsis indicates that successive items must be separated by commas.
[output-source, ...]	Square brackets, in function synopses and a few other contexts, indicate that a syntactic element is optional. Square brackets are not optional, however, when used to delimit a directory name in a VMS file specification or when used to delimit the dimensions of an array in VAX C source code.
sc-specifier ::= auto static [extern] register	In syntax definitions, items appearing on separate lines are mutually exclusive alternatives.
{a b}	Braces surrounding two or more items separated by a vertical bar () indicate a choice; you must choose one of the two syntactic elements.
Δ	A delta symbol is used in some contexts to indicate a single ASCII space character.
switch statement fprintf function auto storage class	In syntax definitions, items appearing in boldface type identify language keywords and the names of VMS and VAX C Run-Time Library functions.

New and Changed Features

The following list documents the features that distinguish VAX C Version 3.0 from previous versions:

- You can decompose loops for parallel processing by specifying the `/PARALLEL` qualifier on the CC command line (see Chapter 1). For more information about parallel-processing programming, see Chapter 3. For more information on parallel-processing debugging, see Appendix E.
- You can improve program performance with automatic inline expansion of function code. For more information, see Chapter 10.
- VAX C now allows you to create separate preprocessor output with the `/PREPROCESS_ONLY` qualifier. For more information, see Chapter 1.
- VAX C now supports new versions of the memory-management functions **malloc**, **calloc**, **free**, **cfree**, and **realloc**. For information about the linking procedure needed to use these functions, see Chapter 1. For information about the functions themselves, see the *VAX C Run-Time Library Reference Manual*.
- You can now specify up to 255 characters for identifier names.
- When you use `/STANDARD=PORTABLE`, the compiler no longer issues portability messages against the inclusion of the `.h` include files provided by VAX C.
- VAX C supports built-in functions that allow more direct access to VAX instructions. For more information, see Chapter 11.
- VAX C offers an additional predefined macro, `CC$parallel`, for use with parallel-processing applications. For more information, see Chapter 11.
- VAX C now supports the VMS License Management Facility. For more information, see the *VAX C Installation Guide*.

The following chapters of this manual are new:

- Chapter 3 (describes parallel-processing features)
- Chapter 4 (the tutorial is now a separate chapter)
- Chapter 11 (all predefined macros and built-in functions are now in a separate chapter)
- Appendix E (describes parallel-processing debugging)

Developing VAX C Programs on VMS Systems





Developing VAX C Programs at the DCL Command Level

This chapter describes the following information about program development on a VMS system:

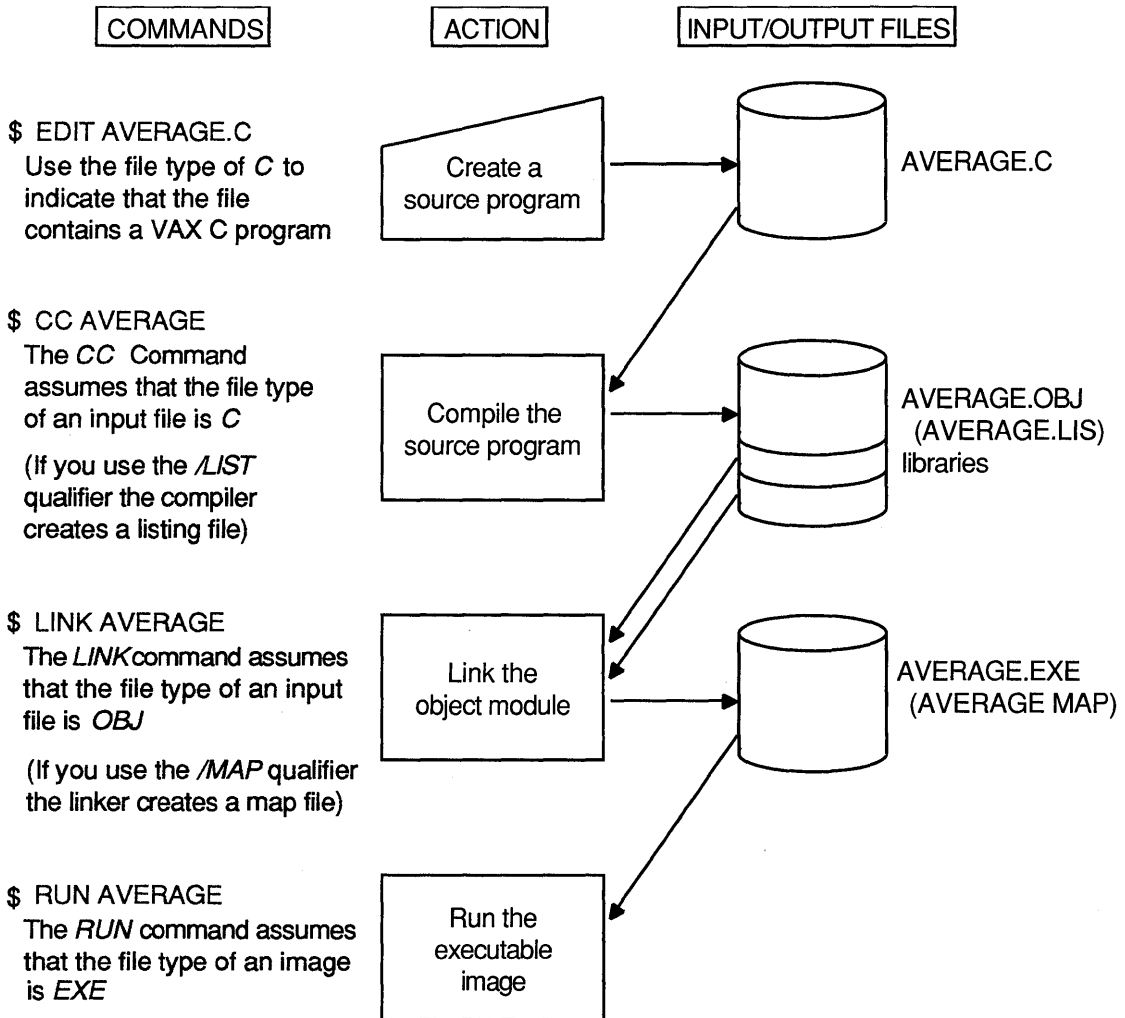
- Overview of Digital Command Language (DCL) commands used for program development (Section 1.1)
- Creating VAX C programs (Section 1.2)
- Compiling VAX C programs (Section 1.3)
- Compilation qualifiers (Section 1.3.2)
- Linking VAX C programs (Section 1.4)
- Linking against object libraries (Section 1.4.5.2)
- Linking against shareable images (Section 1.4.5.3)
- Object module libraries (Section 1.4.5.1)
- Running VAX C programs (Section 1.5)

1.1 DCL Commands for Program Development

This section provides a brief overview of the DCL commands used for program development. The following sections provide more detailed information about these topics.

Figure 1–1 shows the basic steps in VAX C program development.

Figure 1-1: DCL Commands for Developing Programs



ZK-5167-GE

The following example shows each of the commands shown in Figure 1-1 executed in sequence:

```
$ EDIT/TPU AVERAGE.C  
$ CC AVERAGE  
$ LINK AVERAGE  
$ RUN AVERAGE
```

To create a VAX C source program at DCL level, you must invoke a text editor. In the previous example, the VAX Text Processing Utility (VAXTPU) editor is invoked to create the source program AVERAGE.C. You can use another editor, such as VAX EDT or the VAX Language-Sensitive Editor (LSE). (LSE is a product that must be purchased separately; see Appendix C for more information.) C is used as the file type to indicate that you are creating a VAX C source program. C is the conventional file type for all VAX C source programs.

When you compile your program with the CC command, you do not have to specify the file type; by default, VAX C searches for files ending with C.

If your source program compiles successfully, the VAX C compiler creates an object file with the file type OBJ.

However, if the VAX C compiler detects errors in your source program, the system displays each error on your screen and then displays the DCL prompt. You can then reinvoke your text editor to correct each error.

You can include command qualifiers with the CC command. Command qualifiers cause the VAX C compiler to perform additional actions. In the following example, the /LIST qualifier causes the VAX C compiler to produce the listing file AVERAGE.LIS:

```
$ CC/LIST AVERAGE
```

For a complete list and explanation of all the command qualifiers available with the CC command, see Section 1.3.2.

After your program has compiled successfully, invoke the VMS Linker to create an executable image file. The linker uses the object file produced by VAX C as input to produce an executable image file as output. (The executable image is a file containing program code that can be run on the system.)

You can specify command qualifiers with the DCL command LINK. For a complete list and explanation of all the command qualifiers available with the LINK command, see Section 1.4.2.

After producing the executable image file, use the RUN command to execute your program.

1.2 Creating a VAX C Program

To create and modify a VAX C program, you must invoke a text editor. The VMS system provides you with two text editors: VAX EDT (EDT) and the VAX Text Processing Utility (VAXTPU). The following section discusses VAXTPU. See the *VAX EDT Reference Manual* for more information on EDT.

1.2.1 Using VAXTPU

The VAX Text Processing Utility (VAXTPU) is a high-performance, programmable utility. VAXTPU provides two editing interfaces: the Extensible VAX Editor (EVE) and the VAXTPU EDT Keypad Emulator. You can also create your own interfaces.

Like VAX EDT, VAXTPU provides you with an online HELP facility that you can access during your editing session. When you invoke VAXTPU to create a file, a journal file is automatically created. You can use this journal file to recover your edits if the system fails during an editing session. To recover your edits, type the EVE/RECOVER command.

Unlike EDT, VAXTPU provides multiple windows. This feature allows you to view two files on your screen at the same time. VAXTPU also provides you with other advanced features, such as two editing interfaces.

The following sections describe how to use the EVE interface and the EDT Keypad Emulator interface.

1.2.1.1 The EVE Interface

EVE is an interactive text editor that allows you to execute common editing functions using the EVE keypad or to execute more advanced functions by typing commands on the EVE command line. The following command line invokes the EVE editor and creates the file PROG_1.C:

```
$ EDIT/TPU PROG_1.C
```

You can define a global symbol for the EDIT/TPU command by placing a symbol definition in your LOGIN.COM file. For example:

```
$ EVE == "EDIT/TPU"
```

After this command line is executed, you can type EVE at the DCL prompt followed by the name of the file you want to modify or create.

For more information on using the advanced features of EVE, see the *Guide to VMS Text Processing*.

1.3 Compiling a VAX C Program

The VAX C compiler performs the following functions:

- Detects errors in your source program
- Displays each error on your screen or writes the errors to a file
- Generates machine language instructions from the source statements
- Groups these language instructions into an object module for the linker

The following sections discuss the CC command and its qualifiers.

1.3.1 The CC Command

To invoke the VAX C compiler, use the CC command. The CC command has the following format:

```
CC[/qualifier...][ file-spec [/qualifier...]],...
```

/qualifier

Specifies an action to be performed by the compiler on all files or specific files listed. When a qualifier appears directly after the CC command, it affects all the files listed. However, when a qualifier appears after a file specification, it affects only the file that immediately precedes it. However, when files are concatenated, these rules do not apply.

file-spec

Specifies an input source file that contains the program or module to be compiled. You are not required to specify a file type if you give your file a .C file extension; the VAX C compiler adopts the default file type C.

You can include more than one file specification on the same command line by separating the file specifications with either a comma (,) or a plus sign (+). If you separate the file specifications with commas, you can control which source files are affected by each qualifier. In the following example, the VAX C compiler creates an object file for each source file but creates only a listing file for the source files PROG_1 and PROG_3:

```
$ CC /LIST PROG_1, PROG_2/NOLIST, PROG_3
```

If you separate file specifications with plus signs, the VAX C compiler concatenates each of the specified source files and creates one object file and one listing file. In the following example, only one object file is created, PROG_1.OBJ, and only one listing file is created, PROG_1.LIS. Both of these files are named after the first source file in the list, but contain all three modules.

```
$ CC PROG_1 + PROG_2/LIST + PROG_3
```

Any qualifiers specified for a single file within a list of files separated with plus signs affect all the files in the list.

You can specify the name of a text library on the CC command line to compile a source program. A text library is a file that contains text organized into modules indexed by a table. Text libraries have a .TLB default file extension. The modules in the text library have a .TXT file extension, by default.

If it cannot find **#include** modules in libraries specified in the CC command or in the default library defined by the logical name C\$LIBRARY, the VAX C compiler searches the library identified by the following name:

```
SYS$LIBRARY:VAXCDEF.TLB
```

The library VAXCDEF.TLB consists of **#include** modules supplied with VAX C as an option at installation time. In addition, this library contains declarations of values returned by the VMS system services.

Including text modules from the VAXCDEF.TLB library is preferable to including the files in SYS\$LIBRARY with the .H extensions. For example, you can include the Standard I/O definitions in a program with the following **#include** line, which includes the file SYS\$LIBRARY:STDIO.H:

```
#include <stdio.h>
```

You can also use the following line, which includes the text module *stdio* from SYS\$LIBRARY:VAXCDEF.TLB. This method is more efficient.

Including the *stdio* text module is usually quicker than including the STDIO.H file from the SYS\$LIBRARY library directory due to the library indexing system. However, this method is not portable.

```
#include stdio
```

See Section 10.4 for more information on **#include**. See Appendix A for information on definition modules that you can include in your file. See the *VAX C Run-Time Library Reference Manual* for information on the include files that are required to use certain VAX C RTL functions and macros.

1.3.2 The CC Command Qualifiers

The following list shows all the command qualifiers and their defaults available with the CC command. A description of each qualifier follows the list.

Command Qualifiers	Default
/[NO]ANALYSIS_DATA[=file-spec]	/NOANALYSIS_DATA
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG[=(option, . . .)]	/DEBUG=(TRACEBACK,NOINLINE)
/[NO]DEFINE=(identifier[=definition][, . . .])	/NODEFINE
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS
/[NO]G_FLOAT	/NOG_FLOAT
/[NO]INCLUDE_DIRECTORY=(pathname [, . . .])	/NOINCLUDE_DIRECTORY
/LIBRARY	See text.
/[NO]LIST[=file-spec]	/NOLIST (interactive mode) /LIST (batch mode)
/[NO]MACHINE_CODE[=option]	/NOMACHINE_CODE
/[NO]OBJECT[=file-spec]	/OBJECT
/[NO]OPTIMIZE[=option, . . .]	/OPTIMIZE
/[NO]PARALLEL	/NOPARALLEL
/[NO]PRECISION={SINGLE,DOUBLE}	/PRECISION=DOUBLE
/[NO]PREPROCESS_ONLY[=filename]	/NOPREPROCESS
/SHOW[=(option, . . .)]	/SHOW=(NOBRIEF, NODECOMPOSITION, NODICTIONARY, NOEXPANSION, NOINCLUDE, NOINTERMEDIATE, NOSTATISTICS, NOSYMBOLS, NOTRANSLATION, SOURCE, TERMINAL)
/[NO]STANDARD[=(option, . . .)]	/NOSTANDARD
/[NO]UNDEFINE=(identifier[, . . .])	/NOUNDEFINE
/[NO]WARNINGS[=(option, . . .)]	/WARNINGS

You can place command qualifiers either on the CC command line itself or on individual file specifications (with the exception of the /LIBRARY qualifier). If placed on a file specification, the qualifier affects only the compilation of

the specified source file and all subsequent source files in the compilation unit. If placed on the CC command line, the qualifier affects all source files in all compilation units unless it is overridden by a qualifier on an individual file specification.

The rest of this section describes the CC command qualifiers.

/[NO]ANALYSIS_DATA[=file-spec]

Controls whether the compiler generates a file of source-code analysis information. The default file name is the file name of the primary source file; the default file type is .ANA. The .ANA file is reserved for use with DIGITAL layered products. For more information, see Appendix C.

/[NO]CROSS_REFERENCE

Directs the compiler to generate cross-references for variable names. The cross-reference lists each line number in the listing file on which each variable is referenced. This qualifier has no effect unless /LIST and /SHOW=symbols are specified.

The default is /NOCROSS_REFERENCE.

Example 1-1 shows a sample of the type of information placed in the compiler listing when you use /LIST/SHOW=symbols/CROSS_REFERENCE.

Example 1-1: Symbol Cross-References in a Compiler Listing

```
.
.
.
      +-----+
      | Storage Map |
      +-----+
.
.
.
Identifier
  Name      Line      Size      Class      Type and References
-----
main        37                Extern    Function returning
                        def.        long int
                        - No references
timeb       27      10 bytes      Structure tag
                        - Referenced at
                        line 40
.
.
.
```

/[NO]DEBUG[=(option, . . .)]

Requests information to be included in the object module for use by the debugger. Table 1-1 describes the debugger options.

Table 1-1: Debugger Compilation Options

Option	Usage
ALL	Includes symbol table records and traceback records. This is equivalent to /DEBUG=INLINE.
INLINE	Generates debug information to cause a STEP command to STEP/INTO an inlined function call.
NOINLINE	Generates debug information to cause a STEP command to STEP/OVER the inlined function call.
NONE	Does not include any debugging information. This is equivalent to /NODEBUG.

(continued on next page)

Table 1–1 (Cont.): Debugger Compilation Options

Option	Usage
NOTRACEBACK	Does not include traceback records. This option is used to exclude all extraneous information from thoroughly debugged program modules. This option is equivalent to /NODEBUG.
NOSYMBOLS	Includes only traceback records. This is the default if the /DEBUG qualifier is not present on the command line.
SYMBOLS	Includes symbol table records, but <i>not</i> the traceback records.
TRACEBACK	Includes only traceback records. This is the default if the /DEBUG qualifier is not present on the command line.

The default is /DEBUG=(TRACEBACK,NOINLINE).

/[NO]DEFINE=(identifier[=definition][, . . .])

/[NO]UNDEFINE=(identifier[, . . .])

Performs the same functions as the **#define** and **#undefine** preprocessor directives. The /DEFINE qualifier defines a macro to be substituted for every occurrence of a given identifier in the compilation unit or units; /UNDEFINE cancels a previous definition (but not subsequent ones). When both /DEFINE and /UNDEFINE are present in a compilation unit or on the CC command line, /DEFINE is evaluated before /UNDEFINE.

Since the CC command line must be compatible with DCL, the syntax of the /DEFINE and /UNDEFINE qualifiers differs from the syntax of the **#define** and **#undefine** preprocessor directives. The following are differences between the two syntax requirements:

- DCL converts all input to uppercase unless it is enclosed in quotation marks.
- When more than one /DEFINE is present on the CC command line or in a single compilation unit, only the last /DEFINE is used. Similarly, only the last /UNDEFINE is used on the CC command line or the compilation unit.
- DCL accepts only one equal sign as a delimiter, and a space terminates the definition.
- You must use quotation marks to define macro definitions. Within the quotation marks, a delimiter can be either a space or one equal sign, whichever comes first.

The simplest form of a /DEFINE definition is as follows:

```
/DEFINE=true
```

This results in a definition like the one that follows:

```
#define TRUE 1
```

The following example uses the /UNDEFINE qualifier:

```
$ CC/UNDEFINE="TRUE"
```

Since /DEFINE and /UNDEFINE are not part of the source file, they are not associated with a listing line number or source line number. Therefore, when an error occurs in a command-line definition, the message displayed at the terminal does not indicate a line number. In the listing file, these diagnostic messages are placed before the source listing in the order that they were encountered. When the expansion of a definition causes an error at a specific source line in the program, the diagnostics—both at the terminal and in the listing file—are associated with that source line.

A command line containing the /DEFINE and the /UNDEFINE qualifiers can be long. Continuation characters cannot appear within quotes or they will be included in the macro stream. The length of a CC command line cannot exceed the maximum length allowed by DCL.

The /NODEFINE and /NOUNDEFINE qualifiers are provided for compatibility with other DCL qualifiers. You may wish to use these qualifiers to cancel /DEFINE or /UNDEFINE qualifiers that you have specified in a symbol that you use to compile VAX C programs.

The defaults are /NODEFINE and /NOUNDEFINE.

For additional information on the use of these qualifiers, see Section 1.3.2.1.

/[NO]DIAGNOSTICS[=file-spec]

Creates a file containing compiler messages and diagnostic information. The extension .DIA is the default file extension for a diagnostics file. The .DIA file is reserved for use with DIGITAL layered products. For more information, see Appendix C.

The default is /NODIAGNOSTICS.

/[NO]G_FLOAT

Controls the format of floating-point variables. If you do not specify /G_FLOAT on the CC command line, **double** variables are represented in D_floating format. If /G_FLOAT is specified, all variables declared as **double** are represented in G_floating format. (See Section 8.4 for more information on the G_floating format.)

A program compiled with `/G_FLOAT` must also be linked with either the object library `VAXCRTL.G.OLB` or the shareable image `VAXCRTL.G.EXE`. If you are linking against object-module libraries, see Section 1.4.5.2 for information about which libraries to link against and in what order you need to specify these libraries. If you are linking against shareable images, see Section 1.4.5.3.

The default is `/NOG_FLOAT`.

`/[NO]INCLUDE_DIRECTORY=(pathname [, . . .])`

Provides an additional level of search for user-defined include files. Each path-name argument can be either a logical name or a legal directory specification, in quoted form.

The `/INCLUDE_DIRECTORY` qualifier provides the functionality of the `-i` qualifier in CC on ULTRIX. This qualifier allows you to specify additional directories to search for include files. The forms of inclusion affected are the **#include** "file-spec" and **#include** <file-spec> forms. For the quoted form, the order of search is as follows:

1. The directory containing the top-level source file
2. The directories specified in the `/INCLUDE_DIRECTORY` qualifier (if any)
3. The directory or search list of directories specified in the logical name `C$INCLUDE` (if any)

For the bracketed form, the order of search is as follows:

1. The directories specified in the `/INCLUDE_DIRECTORY` qualifier (if any)
2. The directory or search list of directories specified in the logical name `VAXC$INCLUDE` (if any)
3. If `VAXC$INCLUDE` is not defined, then the directory or search list of directories specified by `SY$LIBRARY`

The default is `/NOINCLUDE_DIRECTORY`.

`/LIBRARY`

Indicates that the associated input file is a library containing modules of VAX C source text. If the library specification does not include a file extension, the CC command line assumes the `.TLB` default type. You must join the `/LIBRARY` qualifier with a file specification in a compilation unit using a plus sign (+); you cannot place the qualifier on the CC command line. No matter where you place the `/LIBRARY` qualifier in a compilation unit, all files in the unit may make reference to modules within that library. Consider the following example:

```
$ CC ONE + TWO + THREE/LIBRARY [RETURN]
```

Files ONE.C and TWO.C can contain references to modules in THREE.TLB. Consider the following example:

```
$ CC ONE + TWO + THREE/LIBRARY, FOUR [RETURN]
```

The file FOUR.C cannot contain references to modules in THREE.TLB since FOUR.C is located in a separate compilation unit separated by a comma. The placement of the library file specification does not matter. The following command lines are equivalent:

```
$ CC THREE/LIBRARY + ONE + TWO [RETURN]
$ CC ONE + THREE/LIBRARY + TWO [RETURN]
$ CC ONE + TWO + THREE/LIBRARY [RETURN]
```

/[NO]LIST[=file-spec]

Directs the compiler to produce a listing file containing, by default, a source program listing, a storage map, and a compilation summary. You must specify this qualifier to get any type of listing output. None of the other qualifiers use /LIST by default.

By default, /LIST causes the compiler to create a listing file with the same name as the source file and with the .LIS file extension. If you include a file specification with the /LIST qualifier, the compiler uses that specification to name the listing file.

In interactive mode, the default is /NOLIST. In batch mode, the default is /LIST. See also the descriptions of the qualifiers /[NO]CROSS_REFERENCE, /[NO]MACHINE_CODE, and /SHOW.

/[NO]MACHINE_CODE[=option]

Directs the compiler to list the generated machine code in the listing file. However, the compiler cannot produce any kind of listing file unless you specify /LIST as well.

Several formats exist to list machine code. Table 1–2 shows the options for /MACHINE_CODE.

Table 1–2: /MACHINE_CODE Qualifier Options

Option	Usage
AFTER	Causes the lines of machine code produced during compilation to print after all the source code in the listing.

(continued on next page)

Table 1–2 (Cont.): /MACHINE_CODE Qualifier Options

Option	Usage
BEFORE	Causes lines of machine code produced during compilation to print before any source code in the listing.
INTERSPERSED	Produces a listing consisting of lines of source code followed by the corresponding lines of machine code. This is the default option.

The default is /NOMACHINE_CODE.

/[NO]OBJECT[=file-spec]

Directs the compiler to produce an object module. By default, /OBJECT creates an object module file with the same name as that of the first source file of a compilation unit and with the .OBJ file extension. If you include a file specification with /OBJECT, the compiler uses that specification instead. See Section 1.3.1 for more information about file specifications.

The compiler executes faster if it does not have to produce an object module. Use the /NOOBJECT qualifier when you need only a listing of a program or when you want the compiler to check a file of source text for errors.

The default is /OBJECT.

/OPTIMIZE[=option, . . .]

The /[NO]OPTIMIZE qualifier determines whether VAX C eliminates inefficient code. Table 1–3 presents the /[NO]OPTIMIZE qualifier options.

Table 1–3: /[NO]OPTIMIZE Qualifier Options

Option	Usage
[NO]DISJOINT	Directs the compiler to optimize the generated machine code. For example, the compiler eliminates common subexpressions, removes invariant expressions from loops, collapses arithmetic operations into 3-operand instructions, and places local variables in registers. When debugging VAX C programs, use the /OPTIMIZE=NODISJOINT option if you need minimal optimization; if optimization during debugging is not important, use the /NOOPTIMIZE qualifier.

(continued on next page)

Table 1–3 (Cont.): /[NO]OPTIMIZE Qualifier Options

Option	Usage
/[NO]INLINE	Provides automatic inline expansion of functions that yield optimized code when they are expanded. Whether or not a function is a candidate for inline expansion is based on its size, the number of times it is called, and whether it conforms to the rules specified in Section 10.7.3.1.

The default is /OPTIMIZE, which is the same as /OPTIMIZE=(DISJOINT,INLINE). The /NOOPTIMIZE qualifier turns off the /PARALLEL qualifier.

/[NO]PARALLEL

Specifies whether the compiler should perform dependency analysis on **for** loops in the program and generate optimized code to run on a multiprocessor system.

If you specify /PARALLEL and if you plan on using the memory-management functions **malloc**, **calloc**, **free**, or **cfree**, then you should include the file `stddef.h` in your program and you should link against the proper object library (VAXCPAR.OLB) or shareable image. See Section 1.4.5.2 for information on linking against object-module libraries and Section 1.4.5.3 for information on linking against a shareable image.

The default is /NOPARALLEL. The /NOOPTIMIZE qualifier turns off /PARALLEL.

/[NO]PRECISION= { SINGLE } { DOUBLE }

Directs the compiler to generate code to perform floating-point operations on **float** variables in single or double precision.

Your code may execute faster if it contains **float** variables and is compiled with /PRECISION=SINGLE. However, the results of your floating-point operations will be less precise. See Chapter 8 for more information on floating-point variables.

The default is /PRECISION=DOUBLE.

/[NO]PREPROCESS_ONLY[=filename]

Gives the same functionality as the **-E** qualifier on UNIX C compilers. When it is specified, it causes the compiler to perform only the actions of the preprocessor phase and writes the resulting processed text to a file. No semantic or syntax processing is done. Furthermore, no object file, diagnostic file, listing file, or analysis data file is produced.

If you do not specify a file name for the preprocessor output, the name of the output file defaults to the file name of the input file with a .I file type.

The default is /NOPREPROCESS_ONLY.

/SHOW=[(option, . . .)]

Sets or cancels listing options. You must use the /LIST qualifier with the /SHOW qualifier to use any of the /SHOW options. Table 1–4 presents the /SHOW options.

Table 1–4: /SHOW Qualifier Options

Option	Usage
ALL	Prints all listing information.
[NO]BRIEF	Creates the same listing as the option SYMBOLS except that BRIEF eliminates from the list any identifiers that are not referenced in the program and are not members of a structure or union that is referenced in the program. The /NOBRIEF option is the default.
[NO]DECOMPOSITION	Places a summary of the loops that were decomposed in the listing file. In addition to the /LIST, /OPTIMIZE, and /PARALLEL qualifiers, must be specified for /SHOW=DECOMPOSITION to take effect. The [NO]DECOMPOSITION option is the default.
[NO]DICTIONARY	Places the Common Data Dictionary (CDD) definitions—included in the program with the #dictionary preprocessor directive—into the listing file. These data definitions are marked in the listing file with an uppercase letter D in the listing margin. The NODICTIONARY option is the default.
[NO]EXPANSION	Places final macro expansions in the program listing. When you specify this option, the number of substitutions performed on the line prints next to each line. The NOEXPANSION option is the default.

(continued on next page)

Table 1–4 (Cont.): /SHOW Qualifier Options

Option	Usage
[NO]INCLUDE	Places the contents of #include files and modules in the program listing. The NOINCLUDE option is the default.
[NO]INTERMEDIATE	Places all intermediate and final macro expansions in the program listing. The NOINTERMEDIATE option is the default.
NONE	Creates an empty listing file, with only the header. If you specify this option on a CC command line that contains /LIST and /MACHINE_CODE, the compiler places machine code in the listing file.
[NO]SOURCE	Places the source program statements in the program listing. The SOURCE option is the default.
[NO]STATISTICS	Places compiler performance statistics in the program listing. The NOSTATISTICS option is the default.
[NO]SYMBOLS	Places the symbol table of the compiled program in the program listing. The symbol table includes a list of all functions, the sizes and attributes of all variables referenced in the program, and a program section summary and function definition map. The NOSYMBOLS option is the default.
[NO]TERMINAL	Displays compiler messages to the terminal. The TERMINAL option is the default.
[NO]TRANSLATION	Places into the listing file all UNIX system file specifications that the compiler translates to VMS file specifications using DEC/Shell functions. See the <i>VAX C Run-Time Library Reference Manual</i> for more information on file translation. The NOTRANSFORMATION option is the default.

/[NO]STANDARD[=(option, . . .)]

Directs the compiler to flag certain VAX C specific constructs and VAX C relaxations of conventional C language constructs and rules. For example, the conversions from pointer to integer and back again are subject to more stringent tests when you specify /STANDARD=PORTABLE. If you specify /STANDARD without an option, the default is /STANDARD=PORTABLE. In summary, /STANDARD=PORTABLE causes the compiler to issue warning

messages against coding practices that may not be portable between VAX C and other implementations.

The default is /NOSTANDARD.

/[NO]UNDEFINE=(identifier[, . . .])

See /[NO]DEFINE in this section.

/[NO]WARNINGS[=(option, . . .)]

Controls whether the compiler prints warning diagnostic messages, informational diagnostic messages, neither, or both. The default qualifier, /WARNINGS, causes the compiler to print all diagnostic messages. The /NOWARNINGS qualifier suppresses both the informational and the warning messages.

Table 1-5 presents the two /WARNING qualifier options.

Table 1-5: /WARNINGS Qualifier Options

Option	Usage
NOINFORMATIONALS	Causes the compiler to suppress informational messages.
NOWARNINGS	Causes the compiler to suppress all warning messages.

The informational message, SUMMARY, cannot be suppressed with /NOWARNINGS or /WARNINGS=NOINFORMATIONALS.

The default is /WARNINGS.

1.3.2.1 Using the /DEFINE and /UNDEFINE Qualifiers

This section describes using the /DEFINE and /UNDEFINE qualifiers. Since these qualifiers must follow Digital Command Language (DCL) conventions, their use differs from the use of the **#define** and **#undef** preprocessor control directives.

You must enclose macro definitions in quotation marks. DCL issues a warning message if it encounters a definition of the following form:

```
/DEFINE=funct(a) = a+sin(a)
```

The correct definition is written without spaces, as follows:

```
/DEFINE="funct(a)=a+sin(a)"
```

This definition produces the same results, as follows:

```
#define funct(a) a + sin(a)
```

Within a definition and inside quotes, a delimiter can be either a space or one equal sign, whichever comes first. Consider the following example:

```
$ CC/DEFINE="true=1"
```

This is equivalent to the following:

```
#define true 1
```

Consider the following definition:

```
$ CC/DEFINE="TRUE =1"
```

This definition is equivalent to the following:

```
#define TRUE =1
```

Within the definition and outside quotes, the only allowed delimiter is one equal sign; a space terminates the definition. Consider the following example:

```
$ CC/DEFINE=(maybe=2, "funct(a)=a+sin(a)")
```

These definitions are equivalent to the following:

```
#define MAYBE 2
#define funct(a) a + sin(a)
```

However, the following definitions are not recognized by DCL:

```
$ CC/DEFINE= TRUE
$ CC/DEFINE=(FALSE 0)
```

In the first example, DCL interprets TRUE as a file specification; in the second, DCL flags an invalid value specification.

One equal sign can be passed to the compiler within a single line in one of the following ways:

```
$ CC/DEFINE=(EQU==, "equ =", "equal==")
```

In the first definition, two equal signs are required: the first is removed by DCL as the delimiter; the other is passed to the compiler. In the second example, the space is recognized as a delimiter because the definition is inside quotes. Therefore, only one equal sign is required. In the third definition, the equal sign is used as the delimiter. The compiler removes the first equal sign.

You can pass quotation marks in one of the following ways:

```
$ CC/DEFINE=(QUOTES="\"", "funct(b)=printf("))
```

In both examples, DCL removes the first and last quotation marks before passing the definition to the compiler.

The `/UNDEFINE` qualifier is useful for undefining the predefined VAX C preprocessor constants. For example, if you use a preprocessor constant (such as `vaxc`, `VAXC`, `VAX11c`, or `vms`) to conditionally compile segments of VAX C specific code, you can undefine that constant to see how the portable sections of your program execute. Consider the following program:

```
main()
{
#if vaxc
printf("I'm being compiled with VAX C.");
#else
printf("I'm being compiled on some other compiler.")
#endif
}
```

Output from the program is as follows:

```
$ CC EXAMPLE.C RETURN
$ LINK EXAMPLE.OBJ RETURN
$ RUN EXAMPLE.EXE RETURN
I'm being compiled with VAX C.

$ CC/UNDEFINE="vaxc" EXAMPLE RETURN
$ LINK EXAMPLE.OBJ RETURN
$ RUN EXAMPLE.EXE RETURN
I'm being compiled on some other compiler.
```

1.3.3 Compiler Error Messages

If there are errors in your source file when you compile your program, the VAX C compiler signals these errors and displays diagnostic messages. Reference the diagnostic message, locate the error, and, if necessary, correct the error. Diagnostic messages displayed by VAX C have the following format:

```
%CC-s-ident, message-text
           Listing line number m
           At line number n in name
```

%CC

Is the facility or program name of the VAX C compiler. This portion indicates that the message is being issued by VAX C.

S

Is the severity of the error, represented as follows:

- F Fatal error. The compiler stops executing when a fatal error occurs and does not produce an object module. You must correct the error before you can compile the program.
- E Error. The compiler continues, but does not produce an object module. You must correct the error before you can successfully compile the program.
- W Warning. The compiler produces an object module. It attempts to correct the error in the statement, but you should verify that the compiler's action is acceptable. Otherwise, your program may produce unexpected results.
- I Information. This message usually appears with other messages to inform you of specific actions taken by the compiler. No action is necessary on your part.

ident

Is the message identification. This is a descriptive abbreviation (mnemonic) of the message text.

message-text

Is the compiler's message. In many cases, it consists of more than one line of output. A message generally provides you with enough information to determine the cause of the error so that you can correct it.

Listing line number *m*

Is the integer *m*, which gives you the line number in the listing file where the error occurs. This information is given when you specify the command qualifier /LIST.

At line number *n* in *name*

Is the integer *n*, which gives you the number of the line where the error occurs. The number is relative to the beginning of the file or text library module specified by *name*. You can use the **#line** directive to change both the line number and name that appear in the message.

Appendix B lists the messages produced by the VAX C compiler.

1.4 Linking a VAX C Program

After you compile a VAX C source program or module, use the DCL command LINK to combine your object modules into one executable image, which can then be executed by the VMS system. A source program or module cannot run on the VMS system until it is linked.

When you execute the LINK command, the linker performs the following functions:

- Resolves local and global symbolic references in the object code
- Assigns values to the global symbolic references
- Signals an error message for any unresolved symbolic reference
- Allocates virtual memory space for the executable image

When using the LINK command on development systems, use the /DEBUG qualifier to link your program module. The /DEBUG qualifier appends to the image all the symbol and line number information appended to the object modules plus information on global symbols, and causes the image to run under debugger control when it is executed.

The LINK command produces an executable image by default. However, you can also use the LINK command to obtain shareable images and system images. The /SHAREABLE qualifier directs the linker to produce a shareable image; the /SYSTEM qualifier directs the linker to produce a system image. See Section 1.4.2 for a complete description of these and other LINK command qualifiers.

For a complete discussion of the VMS Linker, see the *VMS Linker Utility Manual*.

1.4.1 The LINK Command

The LINK command has the following format:

```
LINK[command-qualifier]... {file-spec[file-qualifier...]},...
```

/command-qualifier...

Specifies output file options.

file-spec

Specifies the input files to be linked.

/file-qualifier...

Specifies input file options.

If you specify more than one input file, you must separate the input file specifications with a plus sign (+) or a comma (,).

By default, the linker creates an output file with the name of the first input file specified and the file type EXE. If you link more than one file, it is good practice to list the file containing the main program first. Then, the name of your output file will have the same name as your main program module.

The following command line links the object files MAINPROG.OBJ, SUBPROG1.OBJ, and SUBPROG2.OBJ to produce one executable image called MAINPROG.EXE:

```
§ LINK MAINPROG.OBJ, SUBPROG1.OBJ, SUBPROG2.OBJ
```

1.4.2 LINK Command Qualifiers

You can use the LINK command qualifiers to modify the linker's output, as well as to invoke the debugging and traceback facilities. Linker output consists of an image file and an optional map file.

The following list summarizes some of the most commonly used LINK command qualifiers. A brief description of each qualifier follows this list. For a complete list of LINK qualifiers, see the *VMS Linker Utility Manual*.

Command Qualifiers	Default
/BRIEF	See text.
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG	/NODEBUG
/[NO]EXECUTABLE=[file-spec]	/EXECUTABLE=name.EXE
/FULL	See text.
/[NO]MAP	/NOMAP (interactive) /MAP (batch)
/[NO]SHAREABLE[=file-spec]	/NOSHAREABLE
/[NO]TRACEBACK	/TRACEBACK

/BRIEF

Causes the linker to produce a summary of the image's characteristics and a list of contributing modules.

/[NO]CROSS_REFERENCE

Causes the linker to produce cross-reference information for global symbols; /NOCROSS_REFERENCE causes the linker to suppress cross-reference information.

The default is `/NOCROSS_REFERENCE`.

`/[NO]DEBUG`

Causes the linker to include the VMS Debugger in the executable image and generates a symbol table; `/NODEBUG` causes the linker to prevent debugger control of the program.

The default is `/NODEBUG`.

`/[NO]EXECUTABLE [=file-spec]`

Causes the linker to produce an executable image. `/NOEXECUTABLE` suppresses production of an image file.

The default is `/EXECUTABLE`.

`/FULL`

Causes the linker to produce a summary of the image's characteristics, a list of contributing modules, listings of global symbols by name and by value, and a summary of characteristics of image sections in the linked image.

`/[NO]MAP`

Causes the linker to generate a map file; `/NOMAP` suppresses the map.

The default is `/MAP` in batch mode and `/NOMAP` in interactive mode.

`/[NO]SHAREABLE[=file-spec]`

Causes the linker to create a shareable image. `/NOSHAREABLE` generates an executable image.

The default is `/NOSHAREABLE`.

`/[NO]TRACEBACK`

Causes the linker to generate symbolic traceback information when error messages are produced; `NOTRACEBACK` suppresses traceback information.

The default is `/TRACEBACK`.

1.4.3 Linker Input Files

You can specify the object modules to be included in an executable image in any of the following ways:

- Specify input file specifications for the object modules.
If no file type is specified, the linker searches for an object file with the file type OBJ.
- Specify one or more object module library files.

You can specify either the name of an object module library with the `/LIBRARY` qualifier or the names of the object modules contained in an object module library with the `/INCLUDE` qualifier. Section 1.4.5.1 describes the uses of object module libraries.

- Specify an options file.
An options file can contain additional file specifications for the `LINK` command, as well as special linker options. You must use the `/OPTIONS` qualifier to specify an options file. For more information on options files, see the *VMS Linker Utility Manual*.

Table 1–6 shows the default input file types for the linker.

Table 1–6: VMS Linker Default File Types for Input Files

File Type	File
OBJ	Object module
OLB	Library
OPT	Options file

1.4.4 Linker Output Files

When you enter the `LINK` command interactively and do not specify any qualifiers, the linker creates only an executable image file. By default, the resulting image file has the same file name as that of the first object module specified with a file type of `EXE`.

In a batch job, the linker creates both an executable image file and storage map file by default. The default file type for map files is `MAP`.

To specify an alternative name for a map file or image file or to specify an alternative output directory or device, you can include a file specification on the `/MAP` or `/EXECUTABLE` qualifier. In the following example, the `LINK` command creates the image file `[PROJECT.EXE]UPDATE.EXE` and the map file `[PROJECT.MAP]UPDATE.MAP`:

```
$ LINK UPDATE/EXECUTABLE=[PROJECT.EXE]/MAP=[PROJECT.MAP]
```

1.4.5 Linking Against Object Module Libraries and Shareable Images

Linking against object modules (stored in object module libraries) or against shareable images are ways of allowing your program to access data and routines outside of your compilation units. Either the object module libraries and the shareable images can be created by you or they could be ones provided by DIGITAL. To access data in object modules and shareable images, you can use LINK command qualifiers, VMS logical names, and options files.

Also, the VAX C Run-Time Library (RTL) provides two formats for you to choose from: object module libraries or shareable images. Depending on which type of RTL you want to use and on which type of functions you plan on calling from your programs, you need to supply information to the linker that specifies which versions of the functions to access.

When you use the VAX C RTL and its corresponding definition modules (see Appendix A), remember that the VAX C RTL ships with the VMS operating system and the definition modules ship with the VAX C compiler. Since the releases of the compiler and of the operating system are not synchronized, there may be compatibility issues that you need to consider to use the VAX C RTL properly. See the release notes (by typing HELP CC RELEASE_NOTES on the DCL command line) for information that may pertain to this issue.

The following sections discuss these topics in further detail:

- Object module libraries (Section 1.4.5.1)
- Linking against the RTL object libraries (Section 1.4.5.2)
- Linking against the RTL shareable images (Section 1.4.5.3)

1.4.5.1 Object Module Libraries

You can make program modules accessible to other users by storing them in an object module library. To link modules contained in an object module library, use the /INCLUDE qualifier and specify the modules you want to link. In the following example, the LINK command directs the linker to link the subprogram modules EGGPLANT, TOMATO, BROCCOLI, and ONION with the main program module GARDEN:

```
$ LINK GARDEN, VEGGIES/INCLUDE=(EGGPLANT,TOMATO,BROCCOLI,ONION)
```

An object module library can also contain a symbol table with the names of each global symbol in the library, and the name of the module in which they are defined. You specify the name of the object module library containing symbol definitions with the `/LIBRARY` qualifier. When you use the `/LIBRARY` qualifier during a linking operation, the linker searches the specified library for all unresolved references found in the included modules during compilation.

In the following example, the linker uses the library `RACQUETS` to resolve undefined symbols in `BADMINTON`, `TENNIS`, and `RACQUETBALL`:

```
$ LINK BADMINTON, TENNIS, RACQUETBALL, RACQUETS/LIBRARY
```

You can define an object module library to be your default library by using the DCL command `DEFINE LNK$LIBRARY`. The linker searches default user libraries for unresolved references after it searches modules and libraries specified in the `LINK` command. For more information about the `DEFINE` command, see the *VMS DCL Dictionary*.

For more information about object module libraries, see the *VMS Linker Utility Manual*.

1.4.5.2 Linking Against the RTL Object Libraries

Using the object code of the VAX C Run-Time Library (RTL) functions is one of two options (see Section 1.4.5.3 for information on the RTL shareable images). When you choose to use the VAX C RTL as object code, the linker attempts to resolve all references to VAX RTL functions by searching any object module libraries specified on the `LINK` command line. If the linker locates the function code, it places a copy of the code in the program's local program section (psect). If the linker does not locate the function code, it translates the logical name `LNK$LIBRARY_n` to the name of an object library and then searches that library for the code.

If you choose to link against object module libraries and if you want to use any of the VAX C RTL functions, you have to link against the file `SYS$LIBRARY:VAXCTRL.OLB`. Depending on what other VAX C RTL functions you want to use or on other linking requirements, you may have to link against other files in strict order. To use these VAX C RTL functions, define the logicals `LNK$LIBRARY_n` as libraries in the following order, omitting any that you do not need to run your programs:

1. `SYS$LIBRARY:VAXCCURSE.OLB`

Link against this file if you used the Curses Screen Management package of VAX C RTL functions and macros in your compiled program. If you do not need Curses, then do not link against this file.

2. SYS\$LIBRARY:VAXCRTL.G.OLB

Link against this file if you used the /G_FLOAT qualifier on the CC command line. If you do not specify /G_FLOAT, then do not link against this file.

3. SYS\$LIBRARY:VAXCPAR.OLB

Link against this file either to access the parallel-processing versions of the VAX C RTL functions **malloc**, **calloc**, **free**, **cfree**, and **realloc** or to fulfill another linking requirement for parallel processing. (See Section 3.3 for information on linking requirements for parallel processing.)

4. SYS\$LIBRARY:VAXCRTL.OLB

Link against this file to access the VAX C RTL. If you do not use any VAX C RTL functions and if you do not have a VAX C main program, then do not link against this file (or any of the previous files).

If you want to use the regular versions of the VAX C RTL functions (without Curses), then you should define the following logical:

```
$ DEFINE LNK$LIBRARY SYS$LIBRARY:VAXCRTL.OLB RETURN
```

If you need to access all types of VAX C RTL functions and macros, you should define the logical names in the following order:

```
DEFINE LNK$LIBRARY SYS$LIBRARY:VAXCCURSE.OLB RETURN
DEFINE LNK$LIBRARY_1 SYS$LIBRARY:VAXCRTL.G.OLB RETURN
DEFINE LNK$LIBRARY_2 SYS$LIBRARY:VAXCPAR.OLB RETURN
DEFINE LNK$LIBRARY_3 SYS$LIBRARY:VAXCRTL.OLB RETURN
```

If you only need to use Curses, then you should define the logical names in the following order:

```
$ DEFINE LNK$LIBRARY SYS$LIBRARY:VAXCCURSE.OLB RETURN
$ DEFINE LNK$LIBRARY_1 SYS$LIBRARY:VAXCRTL.OLB RETURN
```

If you need to use Curses and G_floating precision in your program, then you should define the logical names in the following order:

```
$ DEFINE LNK$LIBRARY SYS$LIBRARY:VAXCCURSE.OLB RETURN
$ DEFINE LNK$LIBRARY_1 SYS$LIBRARY:VAXCRTL.G.OLB RETURN
$ DEFINE LNK$LIBRARY_2 SYS$LIBRARY:VAXCRTL.OLB RETURN
```

The order of the specified libraries determines which versions of the VAX C RTL functions are found by the linker first. If the linker does not find the function code, or if LNK\$LIBRARY_n is undefined, it assumes that the function is not a VAX C RTL function, and checks the VMS Common Run-Time Procedure Library. These references can be explicit references in your code, or they could be references generated by the compiler to

perform common operations such as input and output, calls to mathematical functions, and so forth.

If the linker cannot resolve the reference by checking the VMS Common Run-Time Procedure Library, it assumes that an error has been made. For more information about Curses, see the *VAX C Run-Time Library Reference Manual*. For more information about the G_floating representation of **double** variables, see Section 8.4. For more information on VAX C support for parallel processing, see Chapter 3.

NOTE

Do *not* use search lists to define the equivalence names for LNK\$LIBRARY_n. The linker will not resolve external references to the VAX C RTL functions in the proper manner.

1.4.5.3 Linking Against the RTL Shareable Images

Using the object code of the VAX C Run-Time Library (RTL) functions is one of two options (see Section 1.4.5.2 for more information). You can also use the VAX C RTL as a shareable image to reduce the space the image takes on the disk and to increase the program execution rate.

When you use the VAX C RTL as a shareable image, you do not receive a copy of the object code in your program's local psect; control is passed, using pointers, from your program to libraries containing the VAX C RTL images where the designated function executes. After execution, control returns to your program. This process has a number of advantages. You significantly reduce the size of a program's executable image, the program's image takes up less disk space, and the program swaps in and out of memory faster due to decreased size.

If you do not use the /G_FLOAT qualifier, then create an options file, OPTIONS_FILE.OPT, containing the following line:

```
SYS$SHARE:VAXCTRL.EXE/SHARE
```

If you *do* use the /G_FLOAT qualifier, then create an options file containing the following line:

```
SYS$SHARE:VAXCTRLG.EXE/SHARE
```

You cannot include the libraries SYS\$SHARE:VAXCTRL.EXE and SYS\$SHARE:VAXCTRLG.EXE in the same options file.

If you have linking requirements for parallel processing (see Section 3.3 for information on compiling and linking requirements), then you also need to link against the VAXCPAR.OLB object module library. To do this, define the following logical name:

```
$ DEFINE LNK$LIBRARY SYSS$LIBRARY:VAXCPAR.OLB RETURN
```

After you define the logical name LNK\$LIBRARY, you can create the options file (described previously) that suits your application.

After you create the appropriate options file, named OPTIONS_FILE.OPT, you can compile and link the program with the following commands:

```
$ CC PROGRAM.C RETURN
$ LINK PROGRAM.OBJ, OPTIONS_FILE/OPT RETURN
```

1.4.6 Linker Error Messages

If the linker detects any errors while linking object modules, it displays messages indicating the cause and severity of the error. If any error or fatal error conditions occur (that is, errors with severities of E or F), the linker does not produce an image file.

The messages produced by the linker are descriptive, and you do not usually need additional information to determine the specific error. Some common errors that occur during linking are as follows:

- An object module has compilation errors.
This occurs when you try to link a module that produced warning or error messages during compilation. You can usually link compiled modules for which the compiler generated messages, but verify that the modules will produce the output you expect.
- The input file has a file type other than OBJ and no file type was specified on the command line.
If you do not specify a file type, the linker searches for a file that has a file type of OBJ by default. If the file is not an object file and you do not identify it with the appropriate file type, the linker signals an error message and does not produce an image file.
- You tried to link a nonexistent module.
The linker signals an error message if you misspell a module name on the command line or if the compilation contains fatal diagnostics.
- A reference to a symbol name remains unresolved.

An error occurs when you omit required module or library names from the command line and the linker cannot locate the definition for a specified global symbol reference. In the following example, a main program module, OCEAN.OBJ, calls the subprogram modules REEF.OBJ, SHELLS.OBJ, and SEAWEED.OBJ, and the following LINK command is executed:

```
$ LINK OCEAN, REEF, SHELLS
```

Because SEAWEED is not linked, the linker signals the following error messages:

```
%LINK-W-NUDFSYMS, 1 undefined symbol
%LINK-I-UDFSYMS,          SEAWEED
%LINK-W-USEUNDEF, module "OCEAN" references undefined symbol "SEAWEED"
%LINK-W-DIAGISUED, completed but with diagnostics
```

If an error occurs when you link modules, you can often correct the error by reentering the command string and specifying the correct modules or libraries. If an error indicates that a program module cannot be located, you may be linking the program with the wrong VAX C RTL.

For a complete list of linker messages, see the *VMS System Messages and Recovery Procedures Reference Volume*.

1.5 Running a VAX C Program

After you link your program, you can use the DCL RUN command to execute it. The RUN command has the following format:

```
RUN [/[NO]DEBUG] file-spec [/[NO]DEBUG]
```

/[NO]DEBUG

Is an optional qualifier. Specify the /DEBUG qualifier to invoke the debugger if the image was not linked with it. You cannot use /DEBUG on images linked with the /NOTRACEBACK qualifier. If the image was linked with the /DEBUG qualifier and you do not want the debugger to prompt you, use the /NODEBUG qualifier. The default action depends on whether the file was linked with the /DEBUG qualifier.

file-spec

Specifies the file you want to run.

The following example executes the image SAMPLE.EXE without invoking the debugger:

```
$ RUN SAMPLE/NODEBUG
```


For more information on debugging programs, see Chapter 2.

During execution, an image can generate a fatal error called an *exception condition*. When an exception condition occurs, the system displays an error message. Run-time errors can also be issued by the operating system or by certain utilities, such as the VMS Sort Utility (SORT).

When an error occurs during the execution of a program, the program is terminated and the VMS condition handler displays one or more messages on the currently defined SYS\$ERROR device.

A message is followed by a traceback. For each module in the image that has traceback information, the condition handler lists the modules that were active when the error occurred, showing the sequence in which the modules were called.

For example, if an integer divide-by-zero condition occurs, a run-time message like the following appears:

```
%SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero
  at PC=00000FC3, PSL=03C00002
```

This message is followed by a traceback message similar to the following:

```
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name  routine name  line    rel  PC      abs PC
A             C             8       00000007  00000FC3
B             main          1408    000002F7  00000B17
```

The information in the traceback message is as follows:

module name

Is the name or names of an image module that was active when the error occurred.

The first module name is that of the module in which the error occurred. Each subsequent line gives the name of the caller of the module named on the previous line. In this example, the modules are A and B; main called C.

routine name

Is the name of the function in the calling sequence.

line

Is the compiler-generated line number of the statement in the source program where the error occurred, or at which the call or reference to the next procedure was made. Line numbers in these messages match those in the listing file.

rel PC

Is the value of the PC (program counter). This value represents the location in the program image at which the error occurred or at which a procedure was called. The location is relative to the virtual memory address that the linker assigned to the code program section of the module indicated by module name.

abs PC

Is the value of the PC in absolute terms; that is, the actual address in virtual memory representing the location at which the error occurred.

Traceback information is available at run time only for modules compiled and linked with the traceback option in effect. The traceback option is in effect by default for both the CC and LINK commands. You may use the CC command qualifier /NODEBUG and the LINK command qualifier /NOTRACEBACK to exclude traceback information. However, traceback information should be excluded only from thoroughly debugged program modules.

Using the VMS Debugger

This chapter is an introduction to using the VMS Debugger (debugger) with VAX C programs and provides the following information:

- An overview of the debugger (Section 2.1)
- Features of the debugger (Section 2.2)
- Information to get you started using the debugger (Section 2.3)
- Debugger support for VAX C (Section 2.4)
- Controlling symbolic references (Section 2.5)
- A sample terminal session that demonstrates using the debugger (Section 2.6)

For complete reference information on the VMS Debugger, see the *VMS Debugger Manual*. Online HELP is available during debugging sessions.

This chapter describes how to debug programs that run in only one process. See Appendix E for more information on debugging programs that take advantage of multiprocess programs.

2.1 Overview

A *debugger* is a tool that helps you locate run-time errors quickly. It is used with a program that has been compiled and linked successfully, but does not run correctly. For example, the output may be obviously wrong, or the program goes into an infinite loop or terminates prematurely. The debugger enables you to observe and manipulate the program's execution interactively so you can locate the point at which the program stopped working correctly.

The VMS Debugger is a *symbolic* debugger, which means that you can refer to program locations by the symbols (names) you used for those locations in your program—the names of variables, routines, labels, and so on. You do not need to use virtual addresses to refer to memory locations.

The debugger recognizes the syntax, expressions, data typing, and other constructs of VAX C, as well as the following VAX-supported languages:

Ada
BASIC
BLISS
COBOL
DIBOL
FORTRAN
MACRO-32
Pascal
PL/I
RPG II
SCAN

If your program is written in more than one language, you can change from one language to another during a debugging session. The current source language determines the format used for entering and displaying data, as well as other features that have language-specific settings (for example, comment characters, operators and operator precedence, and case sensitivity or insensitivity).

By entering debugger commands at your terminal, you can perform the following operations:

- Start, stop, and resume the program's execution
- Trace the execution path of the program
- Monitor selected locations, variables, or events
- Examine and modify the contents of variables, or force events to occur
- Test the effect of some program modifications without having to edit, recompile, and relink the program

These techniques allow you to isolate an error in your code much faster than you could without the debugger.

After you find the error in your program, you can edit the source code and compile, link, and run the corrected version.

2.2 Features of the Debugger

The VMS Debugger provides the following features to help you debug your programs:

- **Online HELP**

Online HELP is available during a debugging session and contains information on all the debugger commands and some selected topics.

- **Source Code Display**

You can display lines of source code during a debugging session.

- **Screen Mode**

You can capture and display various kinds of information in scrollable windows, which can be moved around the screen and resized. Automatically updated source, instruction, and register displays are available. You can selectively direct debugger input, output, and diagnostic messages to displays.

- **Keypad Mode**

When you invoke the debugger, several commonly used debugger command sequences are assigned by default to the keys of the numeric keypad (if you have a VT100, VT52, or LK201 keyboard).

- **Source Editing**

As you find errors during a debugging session, you can use the EDIT command to invoke any editor available on your system. (You first specify the editor you want with the SET EDITOR debugger command).

- **Command Procedures**

The debugger allows you to execute a command procedure to recreate a debugging session, to continue a previous session, or to avoid typing the same debugger commands many times during a debugging session.

- **Symbol Definitions**

You can define your own symbols to represent lengthy commands, address expressions, or values.

- **Initialization Files**

You can create an initialization file containing commands to set your default debugging modes, screen display definitions, keypad key definitions, symbol definitions, and so on. In addition, you may want to have special initialization files for debugging specific programs.

- **Log Files**

You can record the commands you enter during a debugging session and the debugger's responses to those commands in a log file. You can use log files to keep track of your debugging efforts, or you can use them as command procedures in subsequent debugging sessions.

2.3 Getting Started with the Debugger

The following sections explain how to use the debugger with VAX C programs. These sections focus on basic debugger functions to get you started quickly. They also provide any debugger information that is specific to VAX C. For more detailed information that is not specific to a particular language, see the *VMS Debugger Manual*.

2.3.1 Compiling and Linking a Program to Prepare for Debugging

Before using the debugger, you must compile and link your program as explained in this section. The following example shows how to compile and link a VAX C program (consisting of a single compilation unit named INVENTORY) prior to using the debugger:

```
$ CC/DEBUG/NOOPTIMIZE INVENTORY
$ LINK/DEBUG INVENTORY
```

The `/DEBUG` qualifier on the `CC` command line causes the compiler to write the debug symbol records associated with `INVENTORY` into the object module, `INVENTORY.OBJ`. These records allow you to use the names of variables and other symbols declared in `INVENTORY` in debugger commands. (If your program has several compilation units, you must compile each unit that you want to debug with the `/DEBUG` qualifier.)

Use the `/NOOPTIMIZE` qualifier when you compile a program in preparation for debugging. Otherwise, if the object code is optimized (to reduce the size of the program and make it run faster), the contents of some program locations may be inconsistent with what you might expect from viewing the source code. (After debugging the program, recompile it without the `/NOOPTIMIZE` qualifier.)

The `/DEBUG` qualifier on the `LINK` command line causes the linker to include all symbol information that is contained in `INVENTORY.OBJ` in the executable image. This qualifier also causes the VMS image activator to start the debugger at run time. (If your program has several object modules, you may need to specify the other modules in the `LINK` command.)

2.3.2 Starting and Terminating a Debugging Session

You can invoke the debugger in either the *default* or *multiprocess* configuration to debug programs that run in either one or several processes, respectively. The configuration depends on the current value of the logical name `DBG$PROCESS`. Thus, before invoking the debugger, enter the DCL command `SHOW LOGICAL DBG$PROCESS`.

This chapter covers programs that run in only one process. For such programs, `DBG$PROCESS` either should be undefined, as in the following example, or should have the value `DEFAULT`:

```
$ SHOW LOGICAL DBG$PROCESS
%SHOW-S-NOTRAN, no translation for logical name DBG$PROCESS
```

If `DBG$PROCESS` has the value `MULTIPROCESS`, enter the following commands to debug programs that run in only one process (see Appendix E for details on multiprocess debugging):

```
$ DEFINE DBG$PROCESS DEFAULT
```

You can now invoke the debugger by entering the DCL `RUN` command. The following messages then appear on your screen:

```
$ RUN INVENTORY

                                VAX DEBUG Version 5.0

%DEBUG-I-INITIAL, language is C, module set to 'INVENTORY'
DBG>
```

The `INITIAL` message indicates that the debugging session is initialized for a VAX C program and that the name of the main program unit is `INVENTORY`. The `DBG>` prompt indicates that you can now type debugger commands. At this point, if you type the `GO` command, program execution begins and continues until the program is forced to pause or stop (for example, if the program prompts you for input, or an error occurs).

If you have a mixed-language program that includes an Ada package or a program compiled with the `/PARALLEL` qualifier, the following message will appear on your screen instead of the previous one when you invoke the debugger:

```
$ RUN INVENTORY

                                VAX DEBUG Version 5.0

%DEBUG-I-INITIAL, language is C, module set to 'INVENTORY'
%DEBUG-I-NOTATMAIN, type GO to get to start of main program
DBG>
```


The INITIAL message indicates that the debugging session is initialized for a VAX C program and that the name of the main program unit is INVENTORY. The NOTATMAIN message indicates that execution is suspended before the start of the main program, so that you can execute initialization code under debugger control. Typing the GO command places you at the start of the main program. At that point, type the GO command again to start program execution. Execution continues until it is forced to pause or stop (for example, if the program prompts you for input, or an error occurs).

To end a debugging session and return to DCL level, type EXIT or press CTRL/Z:

```
DBG> EXIT
$
```

The following message indicates that your program has completed execution successfully:

```
%DEBUG-I--EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

If you want to continue debugging after seeing this message, type EXIT and start a new debugging session with the DCL RUN command.

2.3.3 Aborting Program Execution or Debugger Commands

If your program loops during a debugging session so that the debugger prompt does not reappear, press CTRL/C. This interrupts program execution and returns you to the prompt. For example:

```
DBG> GO
.
.
.
(infinite loop)
CTRL/C
Interrupt
%DEBUG-W-ABORTED, command aborted by user request
DBG>
```

Do not press CTRL/Y from within a debugging session. Pressing CTRL/Y aborts the session and returns you to the DCL prompt (\$) rather than the debugger prompt.

You can also press CTRL/C to abort the execution of a debugger command. This is useful if a command takes a long time to complete. For example:

```

DBG> EXAMINE/BYTE 1000:101000
1000: 0
1004: 0
1008: 0
1012: 0
1016: 0)
CTRL/C ! Should have typed 1000:1010
%DEBUG-W-ABORTED, command aborted by user request
DBG>

```

If your program has a CTRL/C AST service routine enabled, use the debugger command SET ABORT_KEY to assign the debugger's abort function to another CTRL-key sequence. For example:

```

DBG> SET ABORT_KEY = CTRL_P
DBG> GO
.
.
.
CTRL/P
%DEBUG-W-ABORTED, command aborted by user request
DBG>

```

Note, however, that many CTRL-key sequences have VMS predefined functions, and the SET ABORT_KEY command enables you to override such definitions within the debugging session (see the *VMS DCL Concepts Manual*). Some of the CTRL-key characters not used by the VMS operating system are G, K, N, and P.

2.3.4 Entering Debugger Commands

You can enter debugger commands any time you see the debugger prompt (DBG>). Type the command at the keyboard and press the RETURN key. You can enter several commands on a line by separating the command strings with semicolons (;). As with DCL commands, you can continue a command string on a new line by ending the previous line with a hyphen (-).

You can also use the numeric keypad to enter certain commands. Figure 2-1 shows the predefined key functions. You can also redefine key functions with the DEFINE/KEY command.

Most keypad keys have three predefined functions—DEFAULT, GOLD, and BLUE. (The PF1 key is known as the GOLD key; the PF4 key is known as the BLUE key.) To obtain a key's DEFAULT function, press the key. To obtain its GOLD function, first press the PF1 (GOLD) key, and then the key. To obtain its BLUE function, first press the PF4 (BLUE) key, and then the key.

In Figure 2–1, the DEFAULT, GOLD, and BLUE functions are listed within each key's outline, from top to bottom, respectively. For example, pressing keypad key 0 enters the STEP command; pressing key PF1 and then key 0 enters the STEP/INTO command; pressing key PF4 and then key 0 enters the STEP/OVER command.

Type the command HELP KEYPAD to get help on the keypad key definitions.

Figure 2-1: Debugger Keypad Key Functions

F17 DEFAULT (SCROLL)	F18 MOVE	F19 EXPAND (EXPAND +)	F20 CONTRACT (EXPAND -)
PF1 GOLD GOLD GOLD	PF2 HELP DEFAULT HELP GOLD HELP BLUE	PF3 SET MODE SCREEN SET MODE NOSCR DISP/GENERATE	PF4 BLUE BLUE BLUE
7 DISP SRC,INST,OUT DISP INST,REG,OUT	8 SCROLL/UP SCROLL/TOP SCROLL/UP...	9 DISPLAY next	— DISP next at FS DISP SRC, OUT
4 SCROLL/LEFT SCROLL/LEFT:255 SCROLL/LEFT...	5 EX/SOU .0%PC SHOW CALLS SHOW CALLS 3	6 SCROLL/RIGHT SCROLL/RIGHT:255 SCROLL/RIGHT...	1 GO SEL/INST next
1 EXAMINE EXAM*(prev)	2 SCROLL/DOWN SCROLL/BOTTOM SCROLL/DOWN...	3 SEL/SCROLL next SEL/OUTPUT next SEL/SOURCE next	ENTER
0 STEP STEP/INTO STEP/OVER		* RESET RESET RESET	ENTER

LK201 Keyboard:

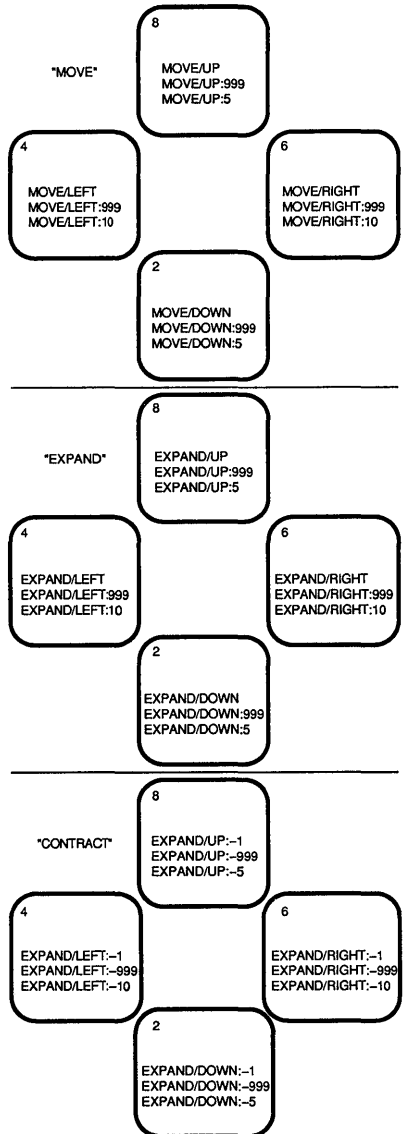
Press
F17
F18
F19
F20

Keys 2,4,6,8
SCROLL
MOVE
EXPAND
CONTRACT

VT-100 Keyboard:

Type
SET KEY:STATE=DEFAULT
SET KEY:STATE=MOVE
SET KEY:STATE=EXPAND
SET KEY:STATE=CONTRACT

Keys 2,4,6,8
SCROLL
MOVE
EXPAND
CONTRACT



ZK-4774-GE

2.3.5 Viewing Your Source Code

The debugger provides two modes for displaying information: noscreen mode and screen mode. By default, when you invoke the debugger, you are in noscreen mode, but you may find that it is easier to view your source code in screen mode. Both modes are briefly described in the following sections.

2.3.5.1 Noscreen Mode

Noscreen mode is the default, line-oriented mode of displaying input and output. To invoke noscreen mode from screen mode, press the keypad key sequence GOLD-PF3. See the sample debugging session in Section 2.6 for a demonstration of noscreen mode.

In noscreen mode, you can use the TYPE command to display one or more source lines. For example, the following command displays line 3 of the module whose code is currently executing:

```
DBG> TYPE 3
module MAIN
    3:  J = 4;
DBG>
```

The display of source lines is independent of program execution. To display source code from a module other than the one whose code is currently executing, use the TYPE command with a path name to specify the module. For example, the following command displays lines 16 through 21 of module TEST:

```
DBG> TYPE TEST\16:21
```

You can also use the EXAMINE/SOURCE command to display the source line for a routine or any other program location that is associated with an instruction.

Note that the debugger also displays source lines automatically when it suspends execution at a breakpoint or watchpoint or after a STEP command, or when a tracepoint is triggered (see Section 2.3.6).

If the debugger cannot locate source lines for display, it enters a diagnostic message. Source lines may not be available for a variety of reasons. For example:

- The module was compiled or linked without the /DEBUG command qualifier.
- Execution is currently suspended within a system or shareable image routine for which no source code is available.

- The module may need to be set with the SET MODULE command. (Section 2.5.1 explains module setting).
- The source file was moved to a different directory after it was compiled (the location of source files is embedded in the object modules). In this case, use the SET SOURCE command to specify the new location.

2.3.5.2 Screen Mode

To invoke screen mode, press keypad key PF3. In screen mode, the debugger splits the screen into three displays named SRC, OUT, and PROMPT, by default. The following example shows how your screen will appear in screen mode:

```
--SRC: module SCOPE---source-scroll-----
  2: *           To be used with F2.C so as to demonstrate the
  3: *           control of modules and setting of scope.
  4:
  5: main()
--> 6: {
      7:   static int i;
      8:   static double f;
      9:   double function2();
     10:   i = 400;
- OUT -output-----

- PROMPT -error-program-prompt-----
DBG>
```

The SRC display, at the top of the screen, shows the source code of the module (compilation unit) where code execution is currently suspended. An arrow in the left column points to the next line to be executed, which corresponds to the current value of the program counter, PC (the PC is a VAX register that contains the address of the next instruction to be executed). The line numbers, which are assigned by the compiler, match those in the listing file.

The OUT display, in the middle of the screen, captures the debugger's output in response to the commands that you enter.

The PROMPT display, at the bottom of the screen, shows the debugger prompt (DBG>), your input, debugger diagnostic messages, and program output.

The SRC and OUT displays can be scrolled to display information beyond the window's edge. Press keypad key 8 to scroll up and keypad key 2 to scroll down. Use keypad key 3 to change the display to be scrolled (by default, the SRC display is scrolled). Scrolling a display does not affect program execution.

In screen mode, if the debugger cannot locate source lines for the program unit where execution is currently suspended, it tries to display source lines in the next routine down on the call stack for which source lines are available. If this is possible, the debugger also enters the following message:

```
%DEBUG-I-SOURCESCOPE, Source lines not available for .O\%PC.  
    Displaying source in a caller of the current routine.
```

In such cases, the arrow in the SRC display identifies the call statement in the calling routine.

2.3.6 Controlling and Monitoring Program Execution

This section discusses the following topics:

- Starting and resuming program execution with the GO command
- Stepping through the program's code with the STEP command
- Determining where execution is currently suspended with the SHOW CALLS command
- Suspending program execution with breakpoints
- Tracing program execution with tracepoints
- Monitoring changes in variables with watchpoints

2.3.6.1 Starting and Resuming Program Execution

There are two debugger commands for starting or resuming program execution: GO and STEP. The GO command starts execution. The STEP command executes a specified number of source lines or instructions.

The GO Command

The GO command starts program execution, which continues until forced to stop. The GO command is used most often in conjunction with breakpoints, tracepoints, and watchpoints (described in Sections 2.3.6.3, 2.3.6.4, and 2.3.6.5). If you set a breakpoint in the path of execution and then enter the GO command, execution is suspended at that breakpoint. If you set a tracepoint, the path of execution through that tracepoint is monitored. If

you set a watchpoint, execution is suspended when the value of the watched variable changes.

You can also use the GO command to test for an exception condition or an infinite loop. If an exception condition that is not handled by your program occurs, the debugger takes control and displays the DBG> prompt so that you can enter commands. If you are using screen mode, the pointer in the source display indicates where execution stopped. You can use the SHOW CALLS command (see Section 2.3.6.2) to identify the currently active routine calls (the call stack).

If an infinite loop occurs, the program does not terminate, so the debugger prompt does not reappear. To obtain the prompt, interrupt execution by pressing CTRL/C (see Section 2.3.3). You can then look at the source display and a SHOW CALLS display to find where execution is suspended.

The STEP Command

The debugger command STEP allows you to execute a specified number of source lines or instructions, or to execute the program to the next instruction of a particular kind, for example, to the next CALL instruction.

By default, the STEP command executes a single source line at a time. In the following example, the STEP command executes one line, reports the action (“stepped to . . . ”), and displays the line number (27) and source code of the next line to be executed:

```
DBG> STEP
stepped to TEST\COUNT\%LINE 27
      27:  x++ ;
DBG>
```

Execution is now suspended at the first machine code instruction for line 27 of the module TEST; line 27 is in COUNT, a routine within the module TEST. TEST\COUNT\%LINE 27 is a path name. The debugger uses path names to refer to symbols. (You do not need to use a path name in referring to a symbol, however, unless the symbol is not unique. If the symbol is not unique, the debugger enters an error message. See Section 2.5.2 for more information on resolving multiply defined symbols.)

The STEP command can execute a number of lines at a time. In the following example, the STEP command executes three lines:

```
DBG> STEP 3
```

Note that only those source lines for which code instructions were generated by the compiler are recognized as executable lines by the debugger. The debugger skips over any other lines—for example, comment lines.

If a line contains more than one statement, the debugger executes all the statements on that line as part of the single step.

You can specify different stepping modes, such as stepping by instruction rather than by line (SET STEP INSTRUCTION). Also, by default, the debugger steps over called routines; execution is not suspended within a called routine, but the routine is executed. Entering the SET STEP INTO command causes the debugger to suspend execution within called routines, as well as within the routine that is currently executing.

2.3.6.2 Determining Where Execution Is Suspended—SHOW CALLS

The debugger command SHOW CALLS is useful when you are unsure where execution is suspended during a debugging session (for example, after a CTRL/C interruption).

The SHOW CALLS command displays a traceback that lists the sequence of calls leading to the routine where execution is currently suspended. For each routine (beginning with the one where execution is suspended), the debugger displays the following information:

- The name of the module that contains the routine
- The name of the routine
- The line number at which the call was made (or at which execution is suspended, in the case of the current routine)
- The corresponding PC addresses (the relative PC address from the start of the routine, and the absolute PC address of the program)

For example:

```
DBG> SHOW CALLS
  module name      routine name      line    rel PC    abs PC
*TEST             PRODUCT          18     00000009 0000063C
*TEST             COUNT            47     00000009 00000647
*MY_PROG          MY_PROG          21     0000000D 00000653
DBG>
```

This example indicates that execution is currently at line 18 of routine PRODUCT (in module TEST), which was called from line 47 of routine COUNT (in module TEST), which was called from line 21 of routine MY_PROG (in module MY_PROG).

2.3.6.3 Suspending Program Execution

The debugger command `SET BREAK` lets you select *breakpoints*, which are locations at which program execution is suspended. When you reach a breakpoint, you can enter commands to check the call stack, examine the current values of variables, and so on.

In the following example, the `SET BREAK` command sets a breakpoint on the procedure `COUNT`. The `GO` command then starts execution. When the procedure `COUNT` is encountered, execution is suspended. The debugger reports that the breakpoint at `COUNT` has been reached (“break at . . .”), displays the source line (54) where execution is suspended, and prompts you for another command. At this breakpoint, you can step through the procedure `COUNT`, using the `STEP` command, and use the debugger command `EXAMINE` (see Section 2.3.7.1) to check on the current values of `X` and `Y`.

```
DBG> SET BREAK COUNT
DBG> GO
.
.
.
break at PROG2\COUNT
    54: {
DBG>
```

When using the `SET BREAK` command, you can specify program locations using various kinds of *address expressions* (for example, line numbers, routine names, instructions, virtual memory addresses, or byte offsets). With high-level languages, you typically use routine names, labels, or line numbers, possibly with path names, to ensure uniqueness.

Specify routine names and labels as they appear in the source code. Line numbers may be derived from either a source code display or a listing file. When specifying a line number, use the prefix `%LINE` or the debugger will interpret the line number as a memory location. For example, the following command sets a breakpoint at line 41 of the module whose code is currently executing; the debugger suspends execution when the PC value is at the start of line 41:

```
DBG> SET BREAK %LINE 41
```

You can only set breakpoints on lines that result in machine code instructions. The debugger warns you if you try to do otherwise (for example, if you try to set a breakpoint on a comment line). To set a breakpoint on a line number in a module other than the one whose code is currently executing, specify the module’s name in a path name as in the following example:

```
DBG> SET BREAK SCREEN_IO\%LINE 58
```

You do not need to specify a particular program location, such as line 58 or COUNT, to set a breakpoint. You can set breakpoints on events, such as exceptions. You can also use the SET BREAK command with the /LINE qualifier (but no parameter) to break on every line, or with the /CALL qualifier to break on every CALL instruction, and so on. For example:

```
DBG> SET BREAK/LINE
DBG> SET BREAK/CALL
```

You can conditionalize a breakpoint (with a WHEN clause) or specify that a list of commands be executed at the breakpoint (with a DO clause). For example, the next command sets a breakpoint on the label loop3. The DO (EXAMINE TEMP) clause causes the value of the variable TEMP to be displayed whenever the breakpoint is triggered.

```
DBG> SET BREAK loop3 DO (EXAMINE TEMP)
DBG> GO
.
.
.
break at COUNTER\loop3
    37:   loop3: for( i = 1; i < 10; i ++ )
COUNTER\TEMP:   284.19
DBG>
```

To display the currently active breakpoints, enter the SHOW BREAK command as follows:

```
DBG> SHOW BREAK
Breakpoint at SCREEN_IO\%LINE 58
Breakpoint at COUNTER\loop3
    do (EXAMINE TEMP)
.
.
.
DBG>
```

If any portion of your program was written in Ada, two breakpoints that are associated with Ada tasking exception events are automatically established when you invoke the debugger. When you enter a SHOW BREAK command under these conditions, the following breakpoints are displayed:

```
DBG> SHOW BREAK
Breakpoint on ADA event "DEPENDENTS_EXCEPTION" for any value
Breakpoint on ADA event "EXCEPTION_TERMINATED" for any value
```

These breakpoints are equivalent to entering the following commands:

```
DBG> SET BREAK/EVENT=DEPENDENTS_EXCEPTION
DBG> SET BREAK/EVENT=EXCEPTION_TERMINATED
```

To cancel a breakpoint, enter the CANCEL BREAK command, specifying the program location or event exactly as you did when setting the breakpoint. The CANCEL BREAK/ALL command cancels all breakpoints.

2.3.6.4 Tracing Program Execution

The debugger command SET TRACE lets you select *tracepoints*, which are locations for tracing the execution of your program without stopping its execution. After setting a tracepoint, you can start execution with the GO command and then monitor the path of execution, checking for unexpected behavior. By setting a tracepoint on a routine, you can also monitor the number of times the routine is called.

As with breakpoints, every time a tracepoint is reached, the debugger enters a message and displays the source line. However, at tracepoints, the program continues executing, and the debugger prompt is not displayed. For example:

```
DBG> SET TRACE COUNT
DBG> GO
.
.
.
trace at PROG2\COUNT
  54: {
.
.
.
```

When using the SET TRACE command, specify address expressions, qualifiers, and optional clauses exactly as with the SET BREAK command.

The /LINE qualifier causes the SET TRACE command to trace every line and is a convenient means of checking the execution path. By default, lines are traced within all called routines, and the currently executing routine. If you do not want to trace through system routines or through routines in shareable images, use the /NOSYSTEM or /NOSHARE qualifiers. For example:

```
DBG> SET TRACE/LINE/NOSYSTEM/NOSHARE
```

The /SILENT qualifier suppresses the trace message and the display of source code. This is useful when you want to use the SET TRACE command to execute a debugger command at the tracepoint. For example:

```
DBG> SET TRACE/SILENT %LINE 83 DO (EXAMINE STATUS)
DBG> GO
.
.
.
SCREEN_IO\CLEAR\STATUS:  0
.
.
.
```

2.3.6.5 Monitoring Changes in Variables

The debugger command SET WATCH lets you set *watchpoints* that will be monitored continuously as your program executes. With high-level languages, you typically set watchpoints on variables that are declared in your program (you can set watchpoints on arbitrary program locations, however). If the program modifies the value of a watched variable, the debugger suspends execution and displays the old and new values.

To set a watchpoint on a variable, specify the variable's name with the SET WATCH command. For example, the following command sets a watchpoint on the variable total:

```
DBG> SET WATCH total
```

Subsequently, every time the program modifies the value of total, the watchpoint is triggered.

The following example shows the effect on program execution when your program modifies the contents of a watched variable:

```
DBG> SET WATCH total
DBG> GO
.
.
.
watch of SCREEN_IO\total at SCREEN_IO\%LINE 13
    13:  total ++;
        old value: 16
        new value: 17
break at SCREEN_IO.%LINE 14
    14:  pop(total);
DBG>
```

In this example, a watchpoint is set on the variable total, and the GO command is entered to start execution. When the value of total changes, execution is suspended. The debugger reports the event (“watch of . . .”) and identifies where total changed (line 13) and the associated source line. The debugger then displays the old and new values and reports that execution has been suspended at the start of the next line (14). (The debugger reports “break at . . .”, but this is not a breakpoint; it is the effect of the watchpoint.) Finally, the debugger prompts for another command.

When a change in a variable occurs at a point other than at the start of a source line, the debugger gives the line number plus the byte offset from the start of the line.

Note that this general technique for setting watchpoints applies to “static” variables. A static variable is associated with the same virtual memory location throughout program execution. In VAX C, variables of the following storage class are statically allocated: **static**, **globaldef**, **globalref**, and **extern**.

A variable that is allocated on the stack or in a register (a “nonstatic” variable) exists only when its defining routine is active (on the call stack). In VAX C nonstatic variables include variables of the storage classes **auto** and **register**. If you try to set a watchpoint on a nonstatic variable when its defining routine is not active, the debugger enters a warning as follows:

```
DBG> SET WATCH Y
%DEBUG-W-SYMNOTACT, nonstatic variable 'Y' is not active
```

A convenient technique for setting a watchpoint on a nonstatic variable is to set a breakpoint on the defining routine, and to specify a DO clause to set the watchpoint whenever execution reaches the breakpoint. In the following example, a watchpoint is set on the nonstatic variable Y in routine COUNTER:

```
DBG> SET BREAK COUNTER DO (SET WATCH Y)
DBG> GO
.
.
.
break at routine MOD4\COUNTER
%DEBUG-I-WPTTRACE, nonstatic watchpoint, tracing every instruction
DBG> SHOW WATCH
watchpoint of MOD4\COUNTER\Y [tracing every instruction]
DBG>
```

The debugger monitors nonstatic watchpoints by tracing every instruction. Because this slows execution speed compared to monitoring static watchpoints, the debugger lets you know when it is monitoring nonstatic watchpoints.

When execution eventually returns to the calling routine, the nonstatic variable is no longer active, so the debugger automatically cancels the watchpoint and enters a message to that effect.

2.3.7 Examining and Manipulating Data

The following sections explain how to use the debugger commands **EXAMINE**, **DEPOSIT**, and **EVALUATE** to display and modify the contents of variables and to evaluate expressions. It also notes restrictions on the use of these commands with VAX C programs.

Before you can examine or deposit into a nonstatic variable (see Section 2.3.6.5), its defining routine must be active (on the call stack).

2.3.7.1 Displaying the Values of Variables

To display the current value of a variable, use the debugger command **EXAMINE**. The **EXAMINE** command has the following form:

```
EXAMINE variable-name
```

The debugger recognizes the compiler-generated data type of the specified variable and retrieves and formats the data accordingly. The following examples show some uses of the **EXAMINE** command:

Examine three integer variables:

```
DBG> EXAMINE WIDTH, LENGTH, AREA
SIZE\WIDTH:    4
SIZE\LENGTH:   7
SIZE\AREA:     28
DBG>
```

Examine a two-dimensional array of integers:

```
DBG> EXAMINE INTEGER_ARRAY
PROG2\INTEGER_ARRAY
[0,0]:    27
[0,1]:    31
[0,2]:    12
[1,0]:    15
[1,1]:    22
[1,2]:    18
DBG>
```

Examine element 4 of a one-dimensional array of characters:

```
DBG> EXAMINE/ASCII CHAR_ARRAY[4]
PROG2\CHAR_ARRAY[4]: 'm'
DBG>
```

You can use the **EXAMINE** command with any kind of address expression, not just a variable name, to display the contents of a program location. The debugger associates certain default data types with untyped locations. You can override the defaults for typed and untyped locations if you want the data to be interpreted and displayed in some other data format.

See Section 2.3.7.3 for a comparison of the **EXAMINE** and **EVALUATE** commands.

2.3.7.2 Changing the Values of Variables

To change the value of a variable, use the debugger command **DEPOSIT**. The **DEPOSIT** command has the following form:

```
DEPOSIT variable-name = value
```

The **DEPOSIT** command is like an assignment statement in VAX C.

In the following examples, the **DEPOSIT** command assigns new values to different variables. The debugger checks that the value assigned, which can be a language expression, is consistent with the data type and dimensional constraints of the variable.

Deposit an integer expression:

```
DBG> DEPOSIT WIDTH = CURRENTWIDTH + 10
```

Deposit element 12 of an array of characters (you cannot deposit an entire array aggregate with a single **DEPOSIT** command, only an element):

```
DBG> DEPOSIT C_ARRAY[12] = 'K'
```

As with the **EXAMINE** command, the **DEPOSIT** command lets you specify any kind of address expression, not just a variable name. You can override the defaults for typed and untyped locations if you want to interpret the data in some other data format.

2.3.7.3 Evaluating Expressions

To evaluate a language expression, use the debugger command **EVALUATE**. The **EVALUATE** command has the following form:

```
EVALUATE language-expression
```

The debugger recognizes the operators and expression syntax of the currently set language. In the following example, the value 45 is assigned to the integer variable **WIDTH**; the **EVALUATE** command then obtains the sum of the current value of **WIDTH** plus 7:

```
DBG> DEPOSIT WIDTH = 45
DBG> EVALUATE WIDTH + 7
52
DBG>
```

Not all VAX C operators can be supported by the debugger, since some can produce side effects that adversely affect debugging. Table 2-1 lists the VAX C operators that are supported in language expressions. Table 2-2 lists the VAX C operators that are not supported by the debugger.

Table 2-1: Supported Operators

Operator(s)	Category
-	Unary arithmetic
+ - * / %	Binary arithmetic
== != > < >= <=	Relational
&& !	Logical
& ^ ? ~	Bitwise logical
<< >>	Shift
sizeof	Compute the size of a scalar
&	Address of
*	Dereference

Table 2-2: Unsupported Operators

Operator(s)	Category
++ --	Pre/post increment/decrement
= += -= *= /=	Assignment
%= = &= ^=	Assignment
?:	Conditional
(type)	Cast

The following example shows the similarity between the EVALUATE and EXAMINE commands. When the expression following the command is a variable name, the value reported by the debugger is the same for either command.

```
DBG> DEPOSIT WIDTH = 45
DBG> EVALUATE WIDTH
45
DBG> EXAMINE WIDTH
SIZE\WIDTH: 45
```

The following example shows an important difference between the EVALUATE and EXAMINE commands:

```
DBG> EVALUATE WIDTH + 7
52
DBG> EXAMINE WIDTH + 7
SIZE\WIDTH: 131584
```

With the EVALUATE command, WIDTH + 7 is interpreted as a language expression, which evaluates to 45 + 7, or 52. With the EXAMINE command,

WIDTH + 7 is interpreted as an address expression: 7 bytes are added to the address of WIDTH, and whatever value is in the resulting address is reported (in this instance, 131584).

2.4 Notes on Debugger Support for VAX C

In general, the debugger supports the data types and operators of VAX C and the other debugger-supported languages. However, there are certain language-specific limitations or other differences. (For information on the supported data types and operators of any of the languages, type HELP LANGUAGE at the DBG> prompt.)

The following sections present VAX C specific debugging examples. These examples show you how to work with VAX C data types and expressions.

2.4.1 Debugger Command-Line Options

VAX C provides a set of debugger options that you can specify to the /DEBUG qualifier to the CC command. These options alter the types of information that the compiler places in the object module for use by the VMS Debugger. The debugger options include using traceback records, using the symbol table, and enabling the debugger to step into inline functions. For information about these options, see the description of the CC command-line qualifiers in Section 1.3.2.

2.4.2 Accessing Scalar Variables

The EXAMINE command displays the scalar variables of any VAX C data type. You reference scalar variables in the case that you declare them, using the VAX C syntax for such references.

Example 2-1 presents the VAX C program SCALARS.C to use in the next sample debugging session.

Example 2-1: Debugging Sample Program SCALARS.C

```
/* SCALARS.C This program defines a large number of      *
 *          variables to demonstrate the effect          *
 *          of the various STEP debugger commands.      */
main()
{
    static float light_speed;          /* Define the variables */
    static double speed_power;
    static unsigned ui;
    static long li;
    static char ch;
    static enum primary { red, yellow, blue } color;
    static int *ptr;

    light_speed = 3.0e10;
    speed_power = 3.1234567890123456789e10;
    ui = -438394;
    li = 790374270;
    ch = 'A';
    color = blue;
    ptr = &li;
}
```

The following debugging session executes SCALARS.EXE and shows the commands used to access variables of scalar data type:

```
DBG> show symbol/type color
data SCALARS\main\color
    enumeration type (primary, 3 elements), size: 4 bytes
```

This command uses the debugger command SHOW SYMBOL/TYPE to display the data type of one variable.

The next commands in this sample debugging session are as follows:

```
DBG> set break %line 22
DBG> go
break at SCALARS\main\%LINE 22
    22: }
```

The commands in the example set a breakpoint before the end of the program and enter a GO command to execute the program up to the breakpoint. These commands allow the variables declared in main to be initialized by the program.

The next command in this sample debugging session is as follows:

```

DBG> examine li, ui, light_speed, speed_power, ch, color, *ptr
SCALARS\main\li:          790374270
SCALARS\main\ui:          4294528902
SCALARS\main\light_speed: 3.0000001E+10
SCALARS\main\speed_power: 31234567890.12346
SCALARS\main\ch:          65
SCALARS\main\color:       blue
*SCALARS\main\ptr:        790374270

```

The **EXAMINE** command directs the debugger to display the contents of the variables listed. The **char** variables are interpreted by the debugger as byte integers, not ASCII characters.

The next command in this sample debugging session is as follows:

```

DBG> examine/ascii ch
SCALARS\main\ch:        "A"

```

To display the contents of **ch** as a character, you must use the **/ASCII** qualifier.

The next command in this sample debugging session is as follows:

```

DBG> deposit/ascii ch = 'z'
DBG> examine/ascii ch
SCALARS\main\ch:        "z"
DBG>

```

The **DEPOSIT** command loads the value 'z' in the variable **ch**; the **EXAMINE** command shows that 'z' has replaced the previous contents of the variable **ch**. Again, use the **/ASCII** qualifier to translate the byte integer into its ASCII equivalent.

2.4.3 Accessing Arrays

With the **EXAMINE** command, you can look at the values in arrays using VAX C syntax for array references. You can examine an entire array by giving the array identifier. You can examine individual elements of the array using the array operators (**[]**). Array elements can have any data type. Remember the differences between pointer arithmetic in VAX C and pointer arithmetic in other languages (see Chapter 8 for more information). Consider the following declaration:

```
int *p;
```

Expression **p+1** is equivalent to the address of **p[1]**; it increments the array by the length specified by 1 multiplied by the length of the data type **int**. Expression **p+1** does not add value 1 to the value of variable **p**: The following debugger commands are equivalent:

```
EVALUATE *(p+1)
EVALUATE p[1]
```

Example 2-2 shows the VAX C program ARRAY.C to use in the next sample debugging session.

Example 2-2: Debugging Sample Program ARRAY.C

```
/* ARRAY.C This program increments an array to          *
 *          demonstrate the access of arrays in VAX C.    */
main()
{
    int i;
    static int arr[10];
    for (i=0; i<10; i++)
        arr[i]=i;
}
```

The following debugging session executes ARRAY.EXE and shows the commands used to access variable arrays:

```
DBG> set br %line 10
DBG> go
break at ARRAY\main\%LINE 10
10: }
```

The commands in Example 2-2 set a breakpoint at the last line in the program and execute the program to that point.

The next command in this sample debugging session is as follows:

```
DBG> examine arr
ARRAY\main\arr
[0]: 0
[1]: 1
[2]: 2
[3]: 3
[4]: 4
[5]: 5
[6]: 6
[7]: 7
[8]: 8
[9]: 9
```

By specifying the variable identifier, you can look at the entire array.

The next command in this sample debugging session is as follows:

```
DBG> examine arr[5]
ARRAY\main\arr[5]:      5
DBG> examine RETURN
ARRAY\main\arr[6]:      6
DBG> examine ^
ARRAY\main\arr[5]:      5
```

Individual elements of the array are examined when you use the bracket operator to specify the subscript of the element. Using the debugger's address reference operator (specified by pressing RETURN) in an EXAMINE command returns the next element of the array. Using the up-arrow address reference operator (^) returns the previous member of the array.

2.4.4 Accessing Character Strings

Character strings are implemented in VAX C as null-terminated ASCII strings (ASCIIZ strings). To examine and deposit data in an entire string, use the /ASCIIZ qualifier (abbreviated /AZ) so that the debugger can interpret the end of the string properly. You can examine and deposit individual characters in the string using the VAX C array subscripting operators ([]). When you examine and deposit individual characters, use the /ASCII qualifier.

Example 2-3 presents the VAX C program STRING.C to use in the next sample debugging session.

Example 2-3: Debugging Sample Program STRING.C

```
/* STRING.C This program establishes a string to      *
 *          demonstrate the access of strings in VAX C. */
main()
{
    static char *s = "vaxie";
    static char **t = &s;
}
```

The following debugging session executes `STRING.EXE` and shows the commands used to manipulate VAX C strings:

```
DBG> step
stepped to STRING\main\%LINE 8
      8: }
DBG> examine/az *s
*STRING\main\s: "vaxie"
DBG> examine/az **t
**STRING\main\t:      "vaxie"
```

The `EXAMINE/AZ` command displays the contents of the character string pointed to by `*s` and `**t`.

The next command in this sample debugging session is as follows:

```
DBG> deposit/az *s = "VAX C"
DBG> examine/az *s, **t
*STRING\main\s: "VAX C"
**STRING\main\t:      "VAX C"
```

The `DEPOSIT/AZ` command deposits a new ASCIIZ string in the variable pointed to by `*s`. The `EXAMINE/AZ` command displays the new contents of the string.

The next command in this sample debugging session is as follows:

```
DBG> examine/ascii s[3]
[3]:  " "
DBG> deposit/ascii s[3] = "-"
DBG> examine/az *s, **t
*STRING\main\s:      "VAX-C"
**STRING\main\t:      "VAX-C"
```

Using array subscripting, you can examine individual characters in the string and deposit new ASCII values at specific locations within the string. When accessing individual members of a string, use the `/ASCII` qualifier. A subsequent `EXAMINE/AZ` command shows the entire string containing the deposited value.

2.4.5 Accessing Structures and Unions

You can examine structures in their entirety or on a member-by-member basis. You can deposit data into structures one member at a time.

You can make references to members of a structure or union by using the usual VAX C syntax for such references. That is, if variable `p` is a pointer to a structure, you can reference member `y` of that structure with the expression `p->y`. If variable `x` refers to the base of the storage allocated for a structure, you can refer to a member of that structure with the `x.y` expression.

To reference members of a structure or union, the debugger follows the VAX C type-checking rules, which follow. For example, in the case of `x.y`, `y` need not be a member of `x`; it is treated as an offset with a type. When such a reference is ambiguous—when there is more than one structure with a member `y`—the debugger attempts to resolve the reference in the following manner. The same rules for resolving the ambiguity of a reference to a member of a structure or union apply to both `x.y` and `p->y`.

- If only one of the members, `y`, belongs in the structure or union, `x`, that is the one that is referenced.
- If only one of the members, `y`, is in the same scope as `x`, then that is the one that is referenced.

You can always give a path name with the reference to `x` to narrow the scope that is used and to resolve the ambiguity. The same path name is used to look up both `x` and `y`.

Example 2–4 shows the VAX C program `STRUCT.C` to use in the next sample debugging session.

The following debugging session executes `STRUCT.EXE` and shows the commands used to access structures and unions:

```
DBG> show symbol * in main
routine STRUCT\main
data STRUCT\main\uv
record component STRUCT\main\<generated_name_0002>.im
record component STRUCT\main\<generated_name_0002>.fm
record component STRUCT\main\<generated_name_0002>.cm
type STRUCT\main\<generated_name_0002>
data STRUCT\main\p
data STRUCT\main\sv
record component STRUCT\main\<generated_name_0001>.im
record component STRUCT\main\<generated_name_0001>.fm
record component STRUCT\main\<generated_name_0001>.cm
record component STRUCT\main\<generated_name_0001>.bf
type STRUCT\main\<generated_name_0001>
```

The `SHOW SYMBOL` command shows the variables contained in the user-defined function `main`.

The next commands in this sample debugging session are as follows:

```
DBG> set break %line 29
DBG> go
break at STRUCT\main\%LINE 29
    29:    uv.im = -24;
```


Example 2-4: Debugging Sample Program STRUCT.C

```
/* STRUCT.C This program defines a structure and union      *
 *           to demonstrate the access of structures and    *
 *           unions in VAX C.                               */
main()
{
    static struct
    {
        int im;
        float fm;
        char cm;
        unsigned bf : 3;
    } sv, *p;

    union
    {
        int im;
        float fm;
        char cm;
    } uv;

    sv.im = -24;
    sv.fm = 3.0e10;
    sv.cm = 'a';
    sv.bf = 7;          /* Binary: 111 */

    p = &sv;

    uv.im = -24;
    uv.fm = 3.0e10;
    uv.cm = 'a';
}

```

Setting a breakpoint at line 29 and entering a GO command allows the program to initialize the variables declared in the structure sv.

The next command in this sample debugging session is as follows:

```
DBG> examine sv
STRUCT\main\sv
  im: -24
  fm: 3.0000001E+10
  cm: 97
  bf: 7

```

An EXAMINE command that gives the name of the structure causes the debugger to display all members of the structure. Note that sv.cm has the **char** data type, which is interpreted by the debugger as a byte integer. The debugger also displays the value of bit fields in decimal.

The next commands in this sample debugging session are as follows:

```
DBG> examine/ascii sv.cm
STRUCT\main\sv.cm:      "a"
DBG> examine/binary sv.bf
STRUCT\main\sv.bf:      111
```

To display the ASCII representation of a **char** data type, you must use the /ASCII qualifier. To display bit fields in their binary representation, you must use the /BINARY qualifier.

The next commands in this sample debugging session are as follows:

```
DBG> deposit sv.im = 99
DBG> deposit sv.fm = 3.14
DBG> deposit/ascii sv.cm = 'z'
DBG> deposit sv.bf = %BIN 010
DBG> examine sv
STRUCT\main\sv
  im: 99
  fm: 3.140000
  cm: 122
  bf: 2
```

You deposit data into a structure one member at a time. To deposit data into a member of type **char**, you can use the /ASCII qualifier and enclose the character in either single or double quotes. To deposit a new binary value in a bit field, use the %BIN keyword.

The next commands in this sample debugging session are as follows:

```
DBG> examine *p
*STRUCT\main\p
  im: 99
  fm: 3.140000
  cm: 122
  bf: 2
DBG> examine/binary p -->bf
STRUCT\main\p -->bf:      010
```

Members of structures (and unions) can also be accessed by pointer, as shown in *p and p -->bf in the previous example.

The next commands in this sample debugging session are as follows:

```
DBG> step
stepped to STRUCT\main\%LINE 30
 30:   uv.fm = 3.0e10;
DBG> examine uv
STRUCT\main\uv
  im: -24
  fm: -1.5485505E+38
  cm: -24
```

A union contains only one member at a time, so the value for uv.im is the only valid value returned by the EXAMINE command; the other values are meaningless.

The next commands in this sample debugging session are as follows:

```
DBG> step
stepped to STRUCT\main\%LINE 31
   31:   uv.cm = 'a';
DBG> examine uv.fm
STRUCT\main\uv.fm:   3.0000001E+10
DBG> step
stepped to STRUCT\main\%LINE 32
   33: }
DBG> examine/ascii uv.cm
STRUCT\main\uv.cm:   "a"
```

This series of STEP and EXAMINE commands shows the content of the union as the different members are assigned values.

Example 2-5 shows the VAX C program ARSTRUCT.C to use in the next sample debugging session.

Example 2-5: Debugging Sample Program ARSTRUCT.C

```
/* ARSTRUCT.C This program contains a structure definition *
 * and a for loop so as to demonstrate the *
 * debugger's support for VAX C operators. */

main()
{
    int    count, i = 1;
    char  c = 'A';

    struct
    {
        int digit;
        char alpha;
    } tbl[27], *p;

    for (count = 0; count <= 26; count++)
    {
        tbl[count].digit = i++;
        tbl[count].alpha = c++;
    }
}
```

The following debugging session executes ARSTRUCT.EXE and shows the use of VAX C expressions on the debugger command line:

```
DBG> set break %line 20 when (count == 2)
DBG> go
break at ARSTRUCT\main\%LINE 20
    20:      }
```

Relational operators can be used in expressions (such as `count == 2` in the preceding example) in a **WHEN** clause to set a conditional breakpoint.

The next commands in this sample debugging session are as follows:

```
DBG> evaluate &tbl
2146736881
DBG> evaluate/address tbl
2146736881
```

The first **EVALUATE** command uses VAX C syntax to refer to the address of a variable. It is equivalent to the second command, which uses the **/ADDRESS** qualifier to obtain the address of the variable. The addresses of these variables may not be the same every time you execute the program if you relink the program.

The next command in this sample debugging session is as follows:

```
DBG> evaluate tbl[2].digit
3
```

Individual members of an aggregate can be evaluated; the debugger returns the value of the member.

The next commands in this sample debugging session are as follows:

```
DBG> evaluate tbl +4
%DEBUG-I-SCALEADD, pointer addition: scale factor of 5 applied to
right argument
2146736901
DBG> examine 2146736901
ARSTRUCT\main\tbl[4].digit:      5
```

When you perform pointer arithmetic, the debugger displays a message indicating the scale factor that has been applied. It then returns the address resulting from the arithmetic operation. A subsequent **EXAMINE** command at that address returns the value of the variable.

The next command in this sample debugging session is as follows:

```
DBG> evaluate tbl[4].digit * 2
10
```

The **EVALUATE** command can perform arithmetic operations on program variables.

The next command in this sample debugging session is as follows:

```
DBG> evaluate 7 % 3
1
```

The `EVALUATE` command can also perform arithmetic calculations that may or may not be related to your program. In effect, it can be used as a calculator that uses VAX C syntax for arithmetic expressions.

The next command in this sample debugging session is as follows:

```
DBG> evaluate count++
%DEBUG-W-SIDEEFFECT, operators with side effects not supported (++, --)
```

The debugger enters a message when you use an unsupported operator.

2.5 Controlling Symbol References

In most cases, the way the debugger handles symbols (variable names, and so on) is transparent to you. However, the following two areas may require action on your part:

- Module setting
- Multiply defined symbols

The following sections describe these two areas.

2.5.1 Module Setting

To facilitate symbol searches, the debugger loads symbol records from the executable image into a run-time symbol table (RST), where they can be accessed efficiently. Unless a symbol record is in the RST, the debugger cannot recognize or properly interpret that symbol.

Because the RST uses memory, the debugger loads it dynamically, anticipating what symbols you might want to reference during execution. The loading process is called *module setting*, because all of the symbol records of a given module are loaded into the RST at one time.

At debugger startup, only the module containing the image transfer address is set. As your program executes, whenever the debugger interrupts execution, it sets the module where execution is suspended. This lets you reference the symbols that should be visible at that location.

If you try to reference a symbol in a module that has not been set, the debugger enters a warning. For example:

```
DBG> EXAMINE K
%DEBUG-W-NOSYMBOL, symbol 'K' is not in symbol table
DBG>
```

You must then use the debugger command `SET MODULE` to manually set the module containing that symbol as follows:

```
DBG> SET MODULE MOD3
DBG> EXAMINE K
MOD3\ROUT2\K: 26
DBG>
```

The debugger command `SHOW MODULE` lists the modules of your program and identifies the modules that have been set.

The dynamic module setting may slow down the debugger as more modules are set. If performance becomes a problem, use the debugger command `CANCEL MODULE` to reduce the number of set modules, or disable dynamic module setting by entering the debugger command `SET MODE NODYNAMIC`. (The `SET MODE DYNAMIC` command enables dynamic module setting.)

2.5.2 Resolving Multiply Defined Symbols

The debugger finds the symbols that you reference in commands according to the following conventions. First, it looks in the PC scope (also known as *scope 0*), according to the scope and visibility rules of the currently set language. This means that the debugger first searches for a symbol within the routine surrounding the current PC value (where execution is currently suspended). If the symbol is not found, the debugger searches the nesting program unit, then its nesting unit, and so on. (The precise order of search depends on the current language and guarantees that the proper declaration of a multiply defined symbol is selected.)

The debugger allows you to reference symbols throughout your program, not just those that are visible at the current PC value, so that you can set breakpoints in arbitrary areas, examine arbitrary variables, and so on. Therefore, if the symbol is not visible in the PC scope, the debugger also searches the scope of the calling routine (if any), then its caller, and so on, until the symbol is found. Symbolically, this search list is denoted $0, 1, 2, \dots, n$, where scope 0 is the PC scope and n is the number of calls in the call stack. Within each scope, the debugger uses the visibility rules of the currently set language to locate symbols.

If the debugger cannot resolve a symbol ambiguity, it enters a warning. For example:

```
DBG> EXAMINE Y
%DEBUG-W-NOUNIQUE, symbol 'Y' is not unique
DBG>
```

You can use a path-name prefix to uniquely specify a declaration of the given symbol. First, use the `SHOW SYMBOL` command to identify all path names associated with the given symbol; then use the desired path name when referencing the symbol. For example:

```
DBG> SHOW SYMBOL Y
data MOD7\ROUT3\BLOCK1\Y
data MOD4\ROUT2\Y
DBG> EXAMINE MOD4\ROUT2\Y
MOD4\ROUT2\Y: 12
DBG>
```

If you need to refer to a particular declaration of `Y` repeatedly, use the `SET SCOPE` command to establish a new default scope for symbol lookup. References to `Y` without a path-name prefix will then specify the declaration of `Y` that is visible in the new scope region. For example:

```
DBG> SET SCOPE MOD4\ROUT2
DBG> EXAMINE Y
MOD4\ROUT2\Y: 12
DBG>
```

To display the current scope for symbol lookup, use the `SHOW SCOPE` command. To restore the default scope, use the debugger command `CANCEL SCOPE`.

2.6 Sample Debugging Session

Example 2-6 shows the VAX C program `POWER.C`, which is to be used in a debugging session. To learn about a larger number of debugger commands, reexecute this program and use a different set of debugger commands.

Example 2-6: Debugging Sample Program POWER.C

```
/* POWER.C This program contains two functions: "main" and *
 * "power." The main function passes a number to *
 * "power", which returns that number raised to the *
 * second power. */

main()
{
    static int i, j;
    int power();

    i = 2;
    j = power(i);
}
power(j)
int j;
{
    return (j * j);
}
```

Example 2-7 shows some of the debugger commands used to evaluate the execution of POWER.C.

Example 2-7: A Sample Debugging Session

```
① $ /CC/DEBUG/OPTIMIZE=NODISJOINT POWER
   $ LINK/DEBUG POWER
   $ RUN POWER

      VAX DEBUG Version 5.n
```

(continued on next page)

Example 2-7 (Cont.): A Sample Debugging Session

```
② %DEBUG-I-INITIAL, language is C, module set to 'POWER'
③ DBG> set break %LINE 13
④ DBG> go
⑤ break at POWER\main\%LINE 13
⑥ 13: j = power(i);
⑦ DBG> step/into
⑧ stepped to routine POWER\power
  16: int j;
  DBG> step
  stepped to POWER\power\%LINE 18
  18: return (j * j)
⑨ DBG> examine J
⑩ %DEBUG-W-NOSYMBOL, symbol 'J' is not in the symbol table
  DBG> examine j
⑪ POWER\power\j: 2
  DBG> step
  stepped to POWER\main\%LINE 13+9
  13: j = power(i);
  DBG> step
  stepped to POWER\main\%LINE 14
  14: }
  DBG> examine j
⑫ POWER\main\j: 4
  DBG> go
⑬ %DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful
  completion'
⑭ DBG> exit
  $
```

Key to Example 2-7:

- ① To execute a program with the debugger, you must compile and link the program with the /DEBUG qualifier. The VAX C compiler compiles the source file with the /DEBUG=TRACEBACK qualifier by default. However, unless you compile your program with the /DEBUG qualifier, you cannot access all of the program's variables. Use the /NOOPTIMIZE qualifier to turn off compiler optimization that may interfere with debugging. If you desire a minimal amount of optimization that will not interfere with your debugging session, use the /OPTIMIZE=NODISJOINT qualifier.
- ② The VMS Image Activator passes control to the debugger on execution of the image. The debugger tells you the current programming language and the name of the object module that contains the main function, or the first function to be executed. Remember that the linker converts the names of object modules to uppercase letters.

- ③ You enter debugger commands at the following prompt:

DBG>

The debugger command SET BREAK defines a point in the program where the debugger must suspend execution. In this example, the SET BREAK command tells the debugger to stop execution before execution of line number 13. After the debugger processes the SET BREAK command, it responds with the debugger prompt.

- ④ The debugger command GO begins execution of the image.
- ⑤ The debugger tells you that it suspended execution of the image at function main in module power. The debugger specifies sections of the program by telling you the object module it is working in, delimited by a backslash character (\), followed by the name of the VAX C function. The linker converted the name of the object module to uppercase letters but the debugger specifies the name of the function exactly as it is found in the source text.
- ⑥ The debugger displays the line of source text where it suspended execution. Refer to the source code listing in Example 2-6 to follow the debugger as it steps through the lines of the program in this interactive debugging example.
- ⑦ The debugger command STEP/INTO executes the first executable line in a function. The command STEP tells the debugger to execute the next line of code, but if the next line of code is a function call, the debugger will not step through the function code unless you use the /INTO qualifier. Use STEP/INTO to step through a user-defined or VAX C RTL function.
- ⑧ When stepping through a function, the debugger specifies line numbers by listing the object module, the VAX C function, a percent sign (%), the identifier LINE, and the line number in the source text. Once again, the debugger delimits all items in the specification with backslash characters (\).
- ⑨ The debugger command EXAMINE displays the contents of a variable.
- ⑩ The debugger does not recognize the variable, J, as existing in the scope of the current module.
- ⑪ The debugger supports the case sensitivity of VAX C variables; variable j exists but variable J does not. Refer to Example 2-6 to review the program variables.

The debugger responds to the EXAMINE command and tells you that the value of the variable is 2.

- ⑫ The value of variable `j` in function `main` is different from the separate variable `j` in function `power`. Function `power` executes properly; it returns the number 2 raised to the second power (4).
- ⑬ Upon completion of execution of the image, the debugger states the status of the execution. In this example, execution is successful.
- ⑭ To enter the `DCL RUN` command to execute the program again, or to do other work outside of the debugger environment, use the debugger command `EXIT` to end the debugging session and to go back to `DCL`.

VAX C Support for Parallel Processing

This chapter describes how to create and to modify programs that run using the VAX C parallel-processing features. This chapter discusses the following topics:

- Overview of parallel processing (Section 3.1)
- Preparing programs for parallel processing (Section 3.2)
- Conditions that inhibit parallel processing (Section 3.3)
- Data-dependency analysis (Section 3.4)
- Rewriting code to resolve dependencies (Section 3.5)
- Storage classes and parallel processing (Section 3.6)
- Decomposition pragmas (Section 3.7)
- Memory-management functions (Section 3.8)
- Tuning issues related to parallel processing (Section 3.9)

See Appendix E for information on debugging programs that use parallel processing.

3.1 Overview of Parallel Processing

Parallel processing involves executing segments of a program concurrently on two or more processors in a multiprocessing system (for example, a VAX 8300 or a VAX 8800; do not confuse these systems with a VAX cluster system). Running programs in parallel on multiple processors, instead of serially on a single processor, can reduce the amount of elapsed time required to run the program. Running programs in parallel, however, consumes more system resources (CPU time and memory) than running serially. Trading off reduced system throughput for reduced elapsed time

is a decision that depends on the application being executed and the environment in which it is being executed.

Not all programs are suitable for parallel execution; some programs are inherently sequential. To achieve maximum benefit, only compute-intensive code sequences should be considered for running in parallel. For example, program segments dealing with arithmetic operations performed on arrays (matrix arithmetic) are generally good candidates for parallel processing. You can identify other compute-intensive code segments using the VAX Performance and Coverage Analyzer (PCA) software product, which can be purchased separately. After isolating code sequences that are candidates for parallel processing, you can then analyze the sequences in detail and make any coding changes that are necessary.

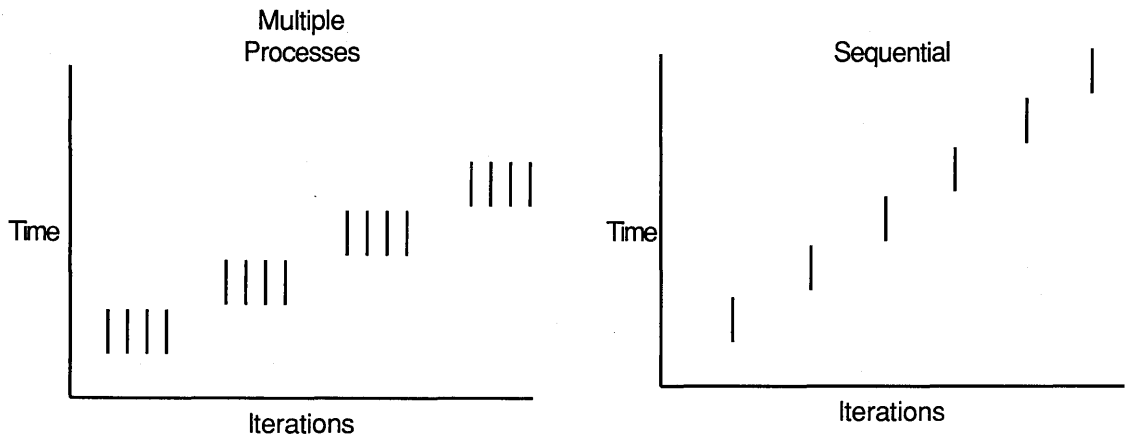
VAX C supports the parallel processing of **for** and **while** loops. (VAX C does not support the parallel processing of **do** loops.) Processing a loop in parallel means that iterations in the loop are divided among processors and are executed concurrently.

Decomposition is the process by which VAX C divides each parallel loop into groups of loop iterations that can be executed concurrently. Figure 3-1 shows parallel and sequential execution of loop iterations over a period of time.

NOTE

Throughout this chapter, loops to be processed in parallel are referred to as parallel loops.

Figure 3-1: Sequential and Parallel Loop Execution Across Time



ZK-6740-GE

When you use the `/PARALLEL` qualifier on the CC command line to compile a program for parallel processing, a VMS Run-Time Library routine sets up the parallel-processing environment. VAX C then tries to decompose each **for** and **while** loop. After compilation, each successfully decomposed loop consists of the following machine code segments:

- Code segment 1 determines the total amount of work to be performed in the loop and sets up global data structures.
- Code segment 2 divides the work into chunks and allocates them to the various processes.
- Code segment 3 is the body of the loop.
- Code segment 4 resets the environment from parallel to sequential processing at the end of each parallel loop.

Each decomposed loop executes in two or more subprocesses, with each subprocess executing a segment of the iterations in the loop. The subprocesses are created during the initialization phase of the program. They are not activated, however, until a parallel **for** or a parallel **while** loop is encountered. When they complete the execution of their portion of the iterations in a parallel loop, they are placed in a wait state until the next parallel loop is encountered.

VAX C does not decompose all **for** and **while** loops. By default, VAX C tries to decompose all **for** loops and tries to decompose **while** loops whose iteration mechanism and number of iterations indicate that the loop is a good candidate for parallel processing. If in a **for** or **while** loop VAX C discovers a possible data dependency, the compiler does not decompose the loop; the compiler executes each iteration of the loop sequentially.

A *data dependency* is a situation that occurs when two or more iterations of the loop depend on a single piece of data. A loop that is a good candidate for parallel processing executes properly and predictably when groups of loop iterations are executed, possibly out of sequence, on separate processors. If a loop iteration depends on the data from a previous or subsequent loop iteration, then the results of the program execution after parallel processing are erroneous or unpredictable.

The following types of data dependencies exist in many parallel-processing applications, although VAX C only checks for loop-carried dependencies:

- **Loop-carried dependency**—A data dependency that occurs when a memory location is both accessed and modified within the same loop. (VAX C checks programs compiled with `/PARALLEL` for loop-carried dependencies.)
- **Loop-independent dependency**—A data dependency that occurs due to the relative positions of two statements in a program. (VAX C does not check programs compiled with `/PARALLEL` for loop-independent dependencies.)
- **Control dependency**—A data dependency introduced by the flow of control in a program. (VAX C does not check programs compiled with `/PARALLEL` for control dependencies.)

VAX C uses algorithms to determine whether data dependencies exist in a loop. VAX C examines loops and their iterations for the following:

- Presence of pointer variables (Section 3.4.3)
- Presence of function calls (Section 3.4.2)
- Existence of two or more iterations making references to the same array element (Section 3.4.1)
- Assigning scalar values and using those values (Section 3.4.4)

VAX C provides mechanisms to use parallel processing and also to suppress the default actions of the compiler during parallel processing. Table 3-1 presents a summary of the VAX C parallel-processing support mechanisms.

Table 3–1: VAX C Parallel-Processing Support Mechanisms

Feature	Description
CC Command-Line Qualifier	
<code>/[NO]PARALLEL</code>	Specifies that a compilation unit is part of a program to be run in parallel. The use of the qualifier determines whether the compiler generates coding structures that are needed to support decomposition of for and while loops.
Decomposition Pragmas	
#pragma ignore_dependency	Specifies to the compiler that a variable that appears to be in conflict is safe to decompose. By default, VAX C does not decompose loops that have two iterations that access the same element. (See Section 3.7.1 for more information about #pragma ignore_dependency .)
#pragma safe_call	Specifies to the compiler that a loop containing a call to the specified function is safe to decompose. By default, VAX C does not decompose loops with function calls. (See Section 3.7.2 for more information on #pragma safe_call .)
#pragma sequential_loop	Specifies to the compiler that the iterations of a for or while loop are to be executed sequentially. By default, VAX C tries to decompose for and while loops for parallel processing. (See Section 3.7.3 for more information on #pragma sequential_loop .)

(continued on next page)

Table 3–1 (Cont.): VAX C Parallel-Processing Support Mechanisms

Feature	Description
Parallel Object Library	
VAXCPAR.OLB	Contains parallel versions of some VAX C Run-Time Library (RTL) functions. You must link against this object module library if your program's main function is written in VAX C, or if your program calls one of the memory-management functions malloc , calloc , free , cfree , or realloc . See Section 3.8 for more information on the memory-management functions. See Section 3.3 for more information on restrictions placed on programs running in parallel. See Section 1.4.5.2 for more information on linking against object module libraries.
Run-Time Environment Logical Names	
FOR\$PROCESSES FOR\$SPIN_WAIT FOR\$STALL_WAIT	Contain values that adjust some aspects of the run-time environment in which you execute your program. (See Section 3.9.1 for more information.)

3.2 Preparing Programs for Parallel Processing

To process a VAX C program for parallel execution, do the following:

1. Use the `/PARALLEL` qualifier on the `CC` command line for the compilation unit that contains the main function or that contains loops that you want to run in parallel.
2. Examine the compiler messages to determine which loops the compiler decomposed and which it did not. Insert an appropriate decomposition pragma (see Section 10.7) to alter the loop-decomposition decisions made by the compiler (if appropriate to your application). If the VAX Language-Sensitive Editor (LSE) is available on your system, you may use its VAX C decomposition support to add decomposition pragmas. See Appendix C for more information about LSE, which is a product that must be purchased separately.
3. If you inserted decomposition pragmas into the program, then recompile it using the `/PARALLEL` qualifier.
4. Link the program. (See Section 3.3 for information about linking restrictions and requirements.)

5. Optionally, define the logical FOR\$PROCESSES to indicate how many subprocesses are created to run your decomposed program. For example, to create four subprocesses to execute your parallel program, enter the following command:

```
$ DEFINE/JOB FOR$PROCESSES 4
```

If you do not specify a value for FOR\$PROCESSES, it defaults to the number of processors in the multiprocessor VAX you are using.

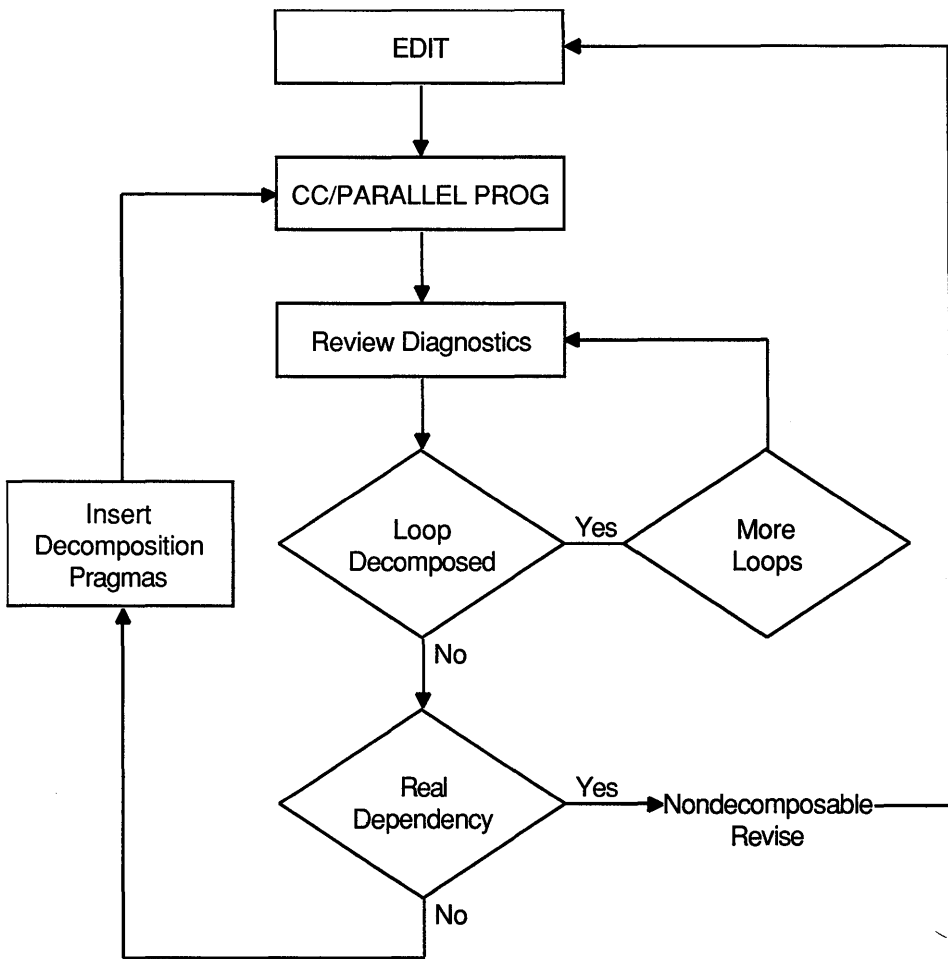
6. Execute the program normally. If you did not define the logical FOR\$PROCESSES, the compiler generates a message when you run your decomposed program.
7. If the program contains run-time errors, use the multiprocessor debugger to debug it. Define the logical DBG\$PROCESS to set up the debugger, as follows:

```
$ DEFINE/JOB DBG$PROCESS MULTIPROCESS
```

See Appendix E for information about debugging decomposed programs.

Figure 3-2 shows a program cycle using decomposition.

Figure 3-2: Program Cycle Using Decomposition



ZK-6739-GE

3.3 Conditions That Inhibit Parallel Processing

You must do the following if you want your program to execute properly in parallel:

- If you want to run some of your VAX C compilation units in parallel, you must use `/PARALLEL` to compile the compilation unit containing the main function, even if the main routine is written in another language. You can use `/PARALLEL` on the other compilation units depending on the needs of your application. If you do not use `/PARALLEL` on the compilation unit containing the main routine, the compiler generates a message.

If you write the main routine of your program in a language that does not support the `/PARALLEL` qualifier, you need to write a shell for the program in a language that does support parallel processing. Then, you must call the main routine from the shell.

See Section 1.3 for more information on compilation units. See Chapter 13 for more information on mixed-language programming.

- If you use `/PARALLEL` when compiling a VAX C compilation unit containing the main function, you must link the program against the `VAXCPAR.OLB` object module library. If you do not use the `/PARALLEL` qualifier on the compilation unit containing the main function or if your main routine runs in parallel but is written in another language, you do not need to link against `VAXCPAR.OLB`.

See Section 1.4.5.2 for information on object module libraries, linking order, and the VAX C Run-Time Library (RTL). See Section 3.8 for information on additional restrictions involving the use of the parallel memory-management functions in the VAX C RTL. See Chapter 13 for more information on mixed-language programming.

VAX C does not decompose loops properly if any of the following conditions exist:

- The loop is a **while** loop and the compiler cannot determine the number of iterations in the loop
- There is a function call in the loop (Section 3.4.2)
- There are pointers used in the loop (Section 3.4.3)
- There exist two or more iterations making references to the same array element (Section 3.4.1)
- There exists a scalar variable that is defined in one loop iteration and referenced in another iteration (Section 3.4.4)
- A **return** or **goto** statement is contained within the loop.

- A label is contained within the loop.
- There are more than 32 variables used within the function containing the loop.
- A **static** or **globaldef** array is referenced or modified within the loop. (See Section 3.6 for more information about storage classes and parallel processing.)
- A **static** or **globaldef** scalar is referenced within the loop. (See Section 3.6 for more information about storage classes and parallel processing.)
- The loop is a **do** loop.
- The loop control variable is not an **[auto]** variable. (See Section 3.6 for more information about storage classes and parallel processing.)
- The loop control variable is a **float** or **double**.
- There is a function call in a loop termination condition.
- The loop is a multiple index loop. You can rewrite this sort of loop as a nested loop to allow decomposition analysis.
- The loop is a nested loop. If a loop contains other loops (nested loops) and all the loops are eligible for decomposition, VAX C only decomposes the outermost loop. Similarly, if your program has a decomposed loop that contains a function call and the function contains a decomposed loop, the loop inside the function does not run in parallel. However, if the function is called from sequential code, the loop inside the function executes in parallel.
- A call to the VAX C RTL function **longjmp**.
- Input or output operations, since they involve function calls.
- Exception or signal handling, since they involve function calls.
- Running your programs from the DEC/Shell. (You must use the DCL command-language interpreter to compile, link, and run parallel programs.)

If you set *errno* in a loop to be decomposed, you must check its value within the decomposed loop. If the program calls a function that sets *errno* from within a decomposed loop and, if it then checks the value of *errno* outside the loop, the value of *errno* at that point reflects the error status of the program code outside the loop, not the error status of the code inside of the loop. Since each subprocess receives its own copy of *errno*, you need to check the value of *errno* periodically inside of the loop.

See Section 3.8 for information about programming restrictions involving the use of the parallel-processing versions of the VAX C RTL memory-management functions **malloc**, **calloc**, **free**, **cfree**, and **realloc**.

3.4 Data-Dependency Analysis

If a data dependency is carried by a **for** or a **while** loop, the result of running it in parallel often varies from sequential execution and varies from one parallel execution to another parallel execution. This unpredictability occurs because loop iterations can be executed out of order when a loop is run in parallel and a loop with a loop-carried data dependency only works correctly when the loop iterations are executed in order.

This section discusses how VAX C analyzes the following calls and references inside each loop body to determine if a loop contains dependencies:

- Array variable references (Section 3.4.1)
- Function calls (Section 3.4.2)
- Pointer variable references (Section 3.4.3)
- Scalar variable references (Section 3.4.4)

3.4.1 Array Variable References

VAX C analyzes all references to arrays in a loop; each array reference is considered in turn. If no element of the array can be modified during execution of the loop, then the values of the array elements are constant in the loop. In this case, the array does not introduce any data dependencies into the loop.

If any array element can be modified within the loop, VAX C must perform further analysis of the loop's references to this array to ensure that no two iterations of the loop reference the same element of the array. In particular, VAX C tries to establish that at least one index of every reference to the array in the loop is distinct in every iteration.

VAX C determines that a simple expression containing the loop's index variable is distinct in each iteration if the expression satisfies the following conditions:

- The index variable appears only once (and is not multiplied by a factor of zero).
- All index-expression values are invariant (unchanging) in the loop.
- All indexed references to the same dimension of the array are identical, except for the constant part.

If an array reference has a constant part, then the compiler computes the distance. If $distance \text{ MOD } step_size = 0$ (where `step_size` is the amount of the increment), then a dependency exists and the compiler does not decompose the loop. Otherwise, the compiler decomposes the loop.

Consider the following example:

```
for (i = 0; i < 100; i++)
{
    for (j = 0; j < 100; j++)
    {
        for (k = 0; k < 100; k++)
        {
            abc[j][j + k] = abc[j + 1][j + k] + i * bcd[i][i];
        }
    }
}
```

VAX C does not decompose the loop on `i` because no index expression of the `abc` array refers to the loop-index variable `i`. The `abc` array is the only variable considered in loop analysis here, because it is the only one modified in the loop.

VAX C does not decompose the loop on `j`, either. While `j` is used in both index expressions of `abc`, the index expression for the first dimension is not identical in all references to `abc`, and the index expression for the second dimension contains a reference to the variable `k`. The variable `k` is not constant during execution of the `j` loop, and this prevents decomposition.

Even though the first two loops are not decomposed, the loop on `k` can be decomposed. The second index of every reference to array `abc` is identical (`j + k`) and contains the loop index variable `k`. The remainder of the expression (`j`) is invariant within the `k` loop. The reference to `bcd[i]` does not prevent decomposition on the `k` loop because it is also an invariant value in the `k` loop.

When loop decomposition is inhibited by an array dependency, VAX C issues a message for each line of code in the loop that references the array. For example, consider the following listing fragment:

```
12  1          for (i = 0; i < 10; i++)
13  1          {
14  2              x[i] = x[i - 1] * pi;
15  2              y[i] = y[i + 1] * pi;
```

Lines 14 and 15 generate messages that indicates that the loop-control variable `i` is contained in an expression that is not invariant. However, the compiler decomposes the following loop without generating messages:

```

12  1  for (i = 0; i < 10; i += 2)
13  1  {
14  2      x[i] = x[i - 1] * pi;
15  2      y[i] = y[i + 1] * pi;

```

VAX C requires that the arrays referenced in a loop be **[extern]** or **[auto]** arrays for the loop to decompose. Arrays that have the **static**, **globaldef**, or **globalref** storage-class specifiers cannot have their storage accessed by multiple processes. If you use an array with one of these storage classes within a loop, you receive a message and loop decomposition is inhibited.

NOTE

In this chapter, the notation **[extern]** refers to any variable declared outside of a function that does not have a **static**, **globaldef**, or **globalref** storage-class specifier. When declaring such a variable, the key word **extern** is optional, and hence, the **[extern]** notation.

Similarly, any variable declared inside a function that is not a function parameter and that has an **auto** or **register** storage-class specifier is an **[auto]** variable. If the variable has no storage-class specifier, **[auto]** is the default.

See Chapter 9 for more information about the **[extern]** and **[auto]** storage-class specifiers.

3.4.2 Function Calls

By default, VAX C does not decompose loops containing function calls. Functions that are called inside a loop can introduce unpredictable behavior in a decomposed loop in the following ways:

- If the function called from within the loop is not reentrant, the parallel execution of several iterations of the function may introduce unpredictable behavior.
A nonreentrant function is any function that cannot have several instances active at once. For example, if a function reads and updates a counter in a static variable, it is not reentrant. In general, any function that uses static data is not reentrant.
- If the function has side effects that introduce data dependencies into the loop, the function may behave unpredictably. For instance, if a function updates a global array that is accessed in the decomposed loop from which it is called, you must examine the function carefully to make certain that no dependencies are introduced by the function call. That

is, the function must not read any memory written by other iterations of the loop, or write any memory read by other iterations of the loop.

- If the loop does not have a predictable flow of control, that is, it does not return normally, decomposition cannot proceed properly. For example, if the function calls the VAX C Run-Time Library (RTL) routine **longjmp**, the function cannot be decomposed.

If you determine that the function contains none of the previous restrictions, you can use the **safe_call** pragma to tell the compiler that it is safe to execute the function in parallel for a given loop. See Section 3.7 for information about the **safe_call** pragma and other decomposition pragmas.

If the **safe_call** pragma does not appear before a loop containing a function call, the compiler issues a message and that loop is not decomposed. However, the compiler may still perform the inline optimization on the function in the loop.

The following section describes the use of *math.h* functions in decomposed loops.

3.4.2.1 math.h Function Calls

By default, the compiler does not decompose loops containing function calls. However, VAX C places global **#pragma safe_call** directives in the *math.h* include file. This allows you to use most of the math functions in the VAX C Run-Time Library (RTL) without inhibiting loop decomposition or without requiring you to use **#pragma safe_call** in your program.

Not all of the math functions are safe to call in your programs. By default, using the following math functions inhibits loop decomposition:

- **frexp**
- **modf**

These math functions introduce possible data dependencies by accepting pointer arguments and by returning additional information by using these arguments. You should not use these functions in loops that you want VAX C to decompose.

If you want to check the value of *errno* as possibly set by one of the math functions, you need to place that check inside the loop to be decomposed. (See Section 3.3 for more information.)

NOTE

If you place the **#include math** directive inside of a function definition, the effect of the **#pragma safe_call** directives is only

local to that function. If you call the math functions in other function definitions, VAX C does not decompose loops in that function. To keep the effect of the *math.h* **safe_call** pragmas global, place the **#include** directive outside function definitions.

See the *VAX C Run-Time Library Reference Manual* for more information on the math functions in the VAX C RTL. See Section 3.7 for more information on the **#pragma safe_call** directive. See Section 10.4 for more information on file inclusion.

3.4.3 Pointer Variable References

Using pointer variables inside a loop can make it difficult to determine whether a data dependency exists within the loop. For instance, in the following example, it is impossible to determine whether the access through the pointer variable *p* introduces a data dependency unless it is known whether *p* points to an element of vector:

```
for (i = 0; i < 127; i++)
{
    vector[i] = *p * pi / sin (x);
}
```

The possible data dependency is clear if an arbitrary element of vector is substituted for **p* in the previous expression:

```
for (i = 0; i < 127; i++)
{
    vector[i] = vector[42] * pi / sin (x);
}
```

If multiple arrays are used in the loop, there is the possibility of a data dependency between *p* and every array used in the loop.

VAX C analyzes references to pointers; if the pointer is initialized so that VAX C can determine the identity of an underlying array, using the pointer in the loop may not prevent decomposition (if it does not introduce data dependencies).

If you are working with arrays, you should use the bracket operators to reference array elements. If you use pointer notation to access array elements, the compiler does not decompose the loop. The compiler decomposes a loop containing the following references to array members:

```
for ( i = 0; i < n; i++ )
    p[i] = q[i];
```

The compiler does not decompose a loop containing the following references to array members, even though this method is functionally equivalent to the one in the last example:

```
for ( i = 0; i < n; i++ )
    *p++ = *q++;
```

If VAX C cannot determine whether a pointer dereference introduces a data dependency, the compiler generates a message and does not decompose the loop. The compiler generates a message for every dereference of the pointer in the loop. If you can determine that using the pointer does not introduce a data dependency, you can use the **ignore_dependency** decomposition pragma to inform the compiler of this. See Section 3.7 for information about decomposition pragmas.

3.4.4 Scalar Variable References

To determine if a loop can be decomposed, VAX C analyzes references to scalar variables within the loop. If a scalar variable in a loop introduces a data dependency, the loop is not decomposed. A scalar can introduce a loop-carried data dependency only if the value of the scalar is defined in one iteration of the loop, and used in another. Since iterations of a decomposed loop have no guaranteed execution order, the iteration that is executed last might vary, which causes the value of the scalar to vary at loop termination.

If a scalar variable is not modified during the execution of a loop, its value can be shared by all iterations of the loop; such scalar references do not prevent decomposition. However, if a scalar variable is modified in the loop, it introduces a loop-carried data dependency when either of the following conditions occur:

- If the value of the scalar variable is defined outside the loop and it is used before it is defined inside the loop.
- If the value of a scalar variable is defined inside the loop and it is used outside the loop.

If either of these two conditions exists, the loop is not decomposed and VAX C generates a message.

Consider the following example:

```

a = b = 1;
c = d = 0;
.
.
.
for (i = 0; i < N; i++)
{
    b = f(i * a);
    if (b < 0)
    {
        d = -b;
        c = c + d;
    }
}
printf("%d\n", d);

```

VAX C does not decompose this loop for two reasons. First, scalar variable `c` is initialized outside the loop, then used in a loop iteration before it is defined inside the loop. Second, the final value of `d` from the last iteration is used outside the loop. The references to variables `a` and `b` are not factors preventing decomposition, because `a` is not modified in the loop, and `b` is defined before it is used in the loop, and `b` is not used after the loop.

VAX C also requires that loop index variables and other scalars modified or used in a loop have the **[auto]** storage class in order for decomposition to proceed. Scalars used in a decomposed loop are placed in registers, to ensure that each process executing an iteration has its own private copy. VAX C only places **[auto]** variables in registers. If you modify a scalar that is not an **[auto]** variable in a loop which VAX C is analyzing for decomposition, you receive a compiler message. Scalars that are read but not modified in the loop must have either the **[auto]** or the **[extern]** storage class, or you receive a compiler message and loop decomposition is inhibited.

Even if the scalars in a loop do not cause a data dependency, they can prevent VAX C from decomposing a loop if there are too many of them. Because VAX C tries to place scalars that are modified within loops in registers, if there are more modified scalars in a loop than there are registers available, the loop is not decomposed. When this occurs, a compiler message is issued.

3.5 Rewriting Code to Resolve Dependencies

The following sections describe three coding techniques that you can use to eliminate data-dependency problems that remove a loop from consideration for parallel processing. The three coding techniques are as follows:

- Loop alignment (Section 3.5.1)

- Code replication (Section 3.5.2)
- Loop distribution (Section 3.5.3)

3.5.1 Loop Alignment

Loop alignment changes loop-carried dependencies to loop-independent dependencies. This method works by changing subscripts so that all references to a given array element occur in the same iteration.

The code in the following **for** loop demonstrates an alignment problem:

```
for (i = 2; i < n; i++);
{
    a[i] = b[i];
    c[i] = a[i + 1];
}
```

The first loop iteration accesses the value in memory location `a[i + 1]` and the next iteration stores another value into that location, referencing it as location `a[i]`.

When the code is executed sequentially, the value in memory location `a[i + 1]` is used before another value is stored into that memory location. This is not true if the code is executed in parallel. For example, if loop iterations 4 and 5 execute in separate processes and iteration 5 executes before iteration 4, the value that iteration 4 accesses from the memory location associated with `a[i + 1]` is the value established by iteration 5 in the memory location associated with `a[i]`.

The way to remedy this dependency is to bring into alignment the two references to the memory location in array `a`, that is, the references to `a[i]` and `a[i + 1]`. You can do this by changing the second assignment statement, as follows:

Original Statement

```
c[i] = a[i + 1];
```

Revised Statement

```
c[i-1] = a[i];
```

The revised statement eliminates the data-dependency problem associated with the previous references to memory locations in array `a`. However, to compensate for the change to the array reference, you may have to adjust the loop control values and add appropriate **if** constructs to achieve the same effect as the original loop.

It is also important to maintain the order in which memory locations are accessed. In this case, memory location `a[i + 1]` in the original **for** loop is used in one iteration and then redefined in the next iteration (as memory location `a[i]`). By aligning the references, each iteration operates on only one memory location and, in the original order of the operations, array `a`'s memory locations are defined before they are used. So, in the revised **for** loop being prepared for parallel processing, the statement performing the use operation must be moved ahead of the statement performing the store operation in order to preserve the original order of these operations.

In the following example, additional changes have to be made to the loop, as follows:

Original for Loop

```
for (i = 2; i < n; i++)
{
    a[i] = b[i];
    c[i] = a[i + 1];
}
```

Revised for Loop

```
for (i = 2; i < n + 1; i++)
{
    if (i > 2)
        c[i - 1] = a[i];
    if (i <= n)
        a[i] = b[i];
}
```

Alternatively, you can compensate for the change to the array reference by distributing certain statements outside the loop, as follows:

Original for Loop

```
for (i=2; i < n; i++)
{
    a[i] = b[i];
    c[i] = a[i + 1];
}
```

Revised for Loop

```
if (n >= 2)
    a[2] = b[2];
for (i = 3; i < n; i++)
{
    c[i - 1] = a[i];
    a[i] = b[i];
}
if (n >= 2)
    c[n] = a[n + 1];
```

If statements are distributed outside the loop, tests must be made to control when those statements are executed. Otherwise, they are always executed and that behavior causes an error when the loop has no iterations.

In addition, when using the loop alignment technique to resolve a data dependency, check to ensure that the coding changes that you make to bring one reference into alignment do not cause previously aligned references to become unaligned.

3.5.2 Code Replication

Code replication entails duplicating certain operations to eliminate a data-dependency problem.

The following example shows a data-dependency problem that can be resolved by code replication:

```
for (i = 2; i <= 100; i++)
{
    a[i] = b[i] + c[i];
    d[i] = a[i] + a[i - 1];
}
```

This example contains a loop-carried dependency between memory locations `a[i]` and `a[i - 1]`. The value at memory location `a[i - 1]` is not predictable because, in some instances, it is not defined in one loop iteration before another loop iteration tries to use it. For example, if iterations 2 through 50 are executing in the main process and iterations 51 through 100 are executing in a separate process, loop iteration 51 may try to use memory location `a[i - 1]` before loop iteration 50 has stored a value in that memory location, referencing it as memory location `a[i]`.

To eliminate this problem, establish the value of `a[i - 1]` in a new memory location and then eliminate the reference to the old memory location, substituting a reference to the duplicated memory location. For example, you can revise the **for** loop, as follows:

Original for Loop

```
for (i = 2; i <= 100; i++)
{
    a[i] = b[i] + c[i];
    d[i] = a[i] + a[i - 1];
}
```

Revised for Loop

```
a[2] = b[2] + c[2];
d[2] = a[2] + a[1];
for (i = 3; i <= 100; i++)
{
    a[i] = b[i] + c[i];
    ta = b[i - 1] + c[i - 1];
    d[i] = a[i] + ta;
}
```

In this situation, you compute the value of memory location `a[i - 1]`, store it into temporary variable `ta`, and replace the reference to `a[i - 1]` with a reference to variable `ta`.

Some of the calculations are pulled out of the loop and the iteration count is modified. This is necessary because the reference to `a[i]` in the original loop used the original value of `a[i]`, not one computed by `b[i] + c[i]`. Using the code replication technique generally requires this type of modification to bring references back into alignment.

3.5.3 Loop Distribution

Loop distribution involves breaking down a loop with data-dependency problems into several loops, one or more of which can be run in parallel. For example, consider the following **for** loop:

```
for (i = 1; i <= 100; i++)
{
    a[i] = a[i - 1] + d[i];
    c[i] = b[i] - a[i];
}
```

This loop contains a data dependency and VAX C cannot run it in parallel without producing unpredictable results. As mentioned in the previous section, if loop iterations 1 through 50 are executing on one processor and loop iterations 51 through 100 are executing on another processor, it is likely that loop iteration 51 will try to access a value in memory location `a[i - 1]` before iteration 50 has executed (and stored the necessary value at that location).

To eliminate this problem, you can distribute the **for** loop. For example, you can revise the **for** loop, as follows:

Original for Loop

```
for (i = 1; i <= 100; i++)
{
    a[i] = a[i - 1] + d[i];
    c[i] = b[i] - a[i];
}
```

Revised for Loop

```
for (i = 1; i <= 100; i++)
    a[i] = a[i - 1] + d[i];

for (i = 1; i <= 100; i++)
    c[i] = b[i] - a[i];
```

Given these changes, the second loop can now be executed in parallel.

3.6 Storage Classes and Parallel Processing

Only variables that are mapped to shared memory can be accessed by multiple processes. Variables that are not mapped to shared memory can inhibit loop decomposition. VAX C automatically maps the following variables to shared memory:

- Any variable that is allocated on the stack, such as an automatic scalar or array variable
- Any scalar variable with the **[extern]** storage-class modifier
- Any variable whose address is passed to a function

Variables that have the **globaldef** or **static** attributes are not mapped to shared memory.

Any memory allocated with the **malloc** function is not shared unless you follow certain requirements for using parallel versions of the memory-management functions of the VAX C RTL. See Section 3.8 for more information.

VAX C automatically aligns **[extern]** variables modified within a decomposed loop on a page boundary. This is necessary in order to place them in shared memory.

However, this page alignment can sometimes cause a linker warning to be generated. If a variable is automatically page aligned in one module because it is accessed in a decomposed loop, but it is not page aligned in other modules, you get a linker warning. This can be safely ignored; if you prefer, you can change the alignment of the variable to be page aligned in all modules by using the **_align** declaration modifier.

3.7 Decomposition Pragmas

In addition to rewriting your code to resolve dependencies, you can place decomposition pragma directives into your programs to override the default actions taken by the compiler. Table 3–2 presents the VAX C decomposition pragmas.

Table 3–2: VAX C Decomposition Pragmas

Pragma	Description
#pragma ignore_dependency	Specifies to the compiler that a variable that appears to be in conflict is safe to decompose. By default, VAX C does not decompose loops that have two iterations that access the same element.
#pragma safe_call	Specifies to the compiler that a loop that contains a call to the specified function is safe to decompose. By default, VAX C does not decompose loops with function calls.
#pragma sequential_loop	Specifies to the compiler that the iterations of a for or while loop should be executed sequentially. By default, VAX C tries to decompose all for and while loops for parallel processing.

For the **ignore_dependency** and **sequential_loop** pragmas, a placement of the pragma affects only the next **for** or **while** loop encountered (regardless if the loop contains a reference to any specified pointer or array variable).

For the **safe_call** pragma, the placement of the pragma determines the scope of the pragma's effect. If you place a **safe_call** pragma outside of all function definitions, the pragma affects all **for** and **while** loops from the position of the pragma to the end of the compilation unit. In this case, the effect of the pragma is global.

If you place the **safe_call** pragma inside a function definition, the pragma affects only the next **for** or **while** loop encountered within that function (regardless if the loop contains a call to the specified function). In this case, the effect of the **safe_call** pragma is local to the enclosing function definition.

If you specify the **ignore_dependency** or **sequential_loop** pragmas or if you specify the **safe_call** pragma inside of a function, remember that the pragma does not affect any loops nested within the next **for** or **while** loop encountered. In this case, you must use a pragma preceding each loop you want VAX C to decompose, including nested loops. If you use the **safe_call** pragma outside of all function definitions, this pragma affects all **for** and **while** loops including those that are nested from the occurrence of the pragma to the end of the compilation unit.

The **#pragma ignore_dependency** and **#pragma safe_call** directives require that you specify one or more identifiers to the directives. If you have more than one specifier, you must separate each identifier with a comma and can optionally enclose the identifiers in one set of parentheses. If you do not use parentheses, you must place one space between the pragma keyword and the identifier list; if you do use parentheses, you do not need this space. For example, the following pragma tells VAX C to decompose a loop containing calls to the functions `func1` and `func2`:

```
#pragma safe_call ( func1, func2 )
```

For the three pragmas that require identifiers, you must make sure that the pragma appears after the declaration of the identifiers (array, pointer, or function names). In this way, the compiler can check to make sure that you are passing identifiers of the correct kind to the three pragmas. Consider the following example:

```
int foo();

#pragma safe_call ( foo )

main()
{
    . . .
}

foo()
{
    . . .
}
```

Once you declare the function `foo`, you can specify `foo` to the **safe_call** pragma. (Once you declare a pointer or array variable, you can specify the identifier to the **ignore_dependency** pragma.) In the previous example, it is safe to call `foo`—in any **for** or **while** loop—from the occurrence of the pragma to the end of the compilation unit.

See Section 10.7 to review the syntax lines for each of the decomposition pragmas. The following sections discuss the use of VAX C decomposition pragmas, as follows:

- **#pragma ignore_dependency** (Section 3.7.1)

- **#pragma safe_call** (Section 3.7.2)
- **#pragma sequential_loop** (Section 3.7.3)

3.7.1 The ignore_dependency Decomposition Pragma

When a loop contains a variable that appears to access the same memory location after two or more iterations, use the **#pragma ignore_dependency** directive. This tells VAX C that the loop is safe to decompose despite its appearance. The **ignore_dependency** pragma must be located after the declarations of any variables that you specify to the pragma. The occurrence of this pragma affects only the next **for** or **while** loop encountered (regardless if the loop contains a reference to the specified array or arrays), excluding any nested loops in this loop.

Example 3–1 is an example of the **#pragma ignore_dependency** directive with a variable that is of type array.

Example 3–1: Using the #pragma ignore_dependency Directive

```
int array[50];
main()
{
    int i;

    /* this loop will get inconsistent results */

    array[0] = 1;
    #pragma ignore_dependency(array)
    for (i = 1; i < 50; i++)
    {
        array[i] = array[i-1] + i;
    }

    #pragma sequential_loop
    for (i = 0; i < 50; i++)
        printf("%d \n",array[i]);
}
```

In Example 3–2, the value of the pointed-to object does not change throughout the execution of the program. Therefore, the order in which the loop iterations execute does not matter. It is safe to decompose the loop that contains the pointer variable `*aa`.

Example 3–2: Using the `#pragma ignore_dependency` Directive

```
double a[100],x;
double *aa;

main()
{
    int i;

    init();

    /* This initializes the contents of the array to 1.0. */
    #pragma ignore_dependency aa)
    for (i = 0; i < 100; i++)
        a[i] = *aa;

    #pragma sequential_loop
    for (i = 0; i < 50; ++i)
        printf( "%f \n", a[i]);
}

init()
{
    x = 1.0;
    aa = &x;
}
```

If you have a loop that contains a variable that is part of an address computation, you must insert the `ignore_dependency` pragma in order for VAX C to decompose the loop.

3.7.2 The `safe_call` Decomposition Pragma

To inform VAX C that it is safe to decompose a loop containing calls to one or more functions, use the `#pragma safe_call` directive. (See Section 3.4.2 for information about function calls and data dependency.) The `safe_call` pragma must be located after the declarations of any functions that you specify to the pragma. If you specify the `#pragma safe_call` directive at the top of your compilation unit and outside of all function definitions, the compiler recognizes that all loops in the compilation unit containing calls to the specified functions are to be decomposed (as long as the loops contain no other data dependencies).

Do not specify a function in a **safe_call** pragma if the following conditions are true about the function:

- It is not reentrant.
- It has side effects that introduce dependencies.
- It uses the VAX C Run-Time Library (RTL) routine **longjmp**, or otherwise modifies the normal flow of control.
- It changes the process in some way.
- It takes an address as an argument, and the address points to memory that is not shared.

In Example 3-3, the first **#pragma safe_call** directive is over the block of the *i* loop, which contains the *j* and *k* loops. However, only the *i* loop is affected by the pragma. The second pragma covers the *j* loop, and the third pragma covers the *k* loop. (Using the **safe_call** outside of the function definition in this example would affect all loops from the occurrence of the pragma to the end of the compilation unit.)

Example 3-3: Using the #pragma safe_call Directive

```
#define pi 3.14259
float a[100][100][100], b[50][50][50];

main()
{
    int i,j,k;

#pragma safe_call(func_a) /* Only affects i loop. */
    for (i = 1; i < 100; i++)
    {
        /* Next line gets a message on func_b and func_c. */
        a[i][i][i] = func_b(pi) * func_a(pi) / func_c(pi);
    }

#pragma safe_call(func_b)
    for( j = i; j < 100; ++j)
    {
        /* Next line gets a message on func_a and func_c. */
        a[i][j][i] = func_b(pi) * func_a(pi) / func_c(pi);
    }
}
```

(continued on next page)

Example 3–3 (Cont.): Using the `#pragma safe_call` Directive

```
#pragma safe_call(func_a,func_b,func_c)
/* This loop will be decomposed, since it is the
   only one that contains all safe calls. */
for (k = j; k < 100; k = k + 1)
{
    a[i][j][k] = func_b(pi) * func_a(pi) / func_c(pi);
}
}
```

In Example 3–3, VAX C decomposes the `k` loop, but the compiler issues messages against the statement in the `k` loop. However, these messages apply only to the analysis of the `i` and `j` loops, not the `k` loop. Using the last specified `safe_call` pragma outside of the function definition in this example causes all three loops to be candidates for decomposition.

3.7.3 The `sequential_loop` Decomposition Pragma

To inform VAX C that the iterations of a `for` or `while` loop are to execute sequentially, use the `#pragma sequential_loop` directive. (By default, VAX C tries to decompose all `for` and `while` loops.) This pragma shuts off all decomposition analysis and prevents most decomposition diagnostics from being generated for the loop. The occurrence of this pragma affects only the next `for` or `while` loop encountered, excluding any nested loops in this loop.

Example 3–4 presents an example of using the `#pragma sequential_loop` directive to tell the compiler that the next encountered `for` loop requires sequential execution.

A loop using an I/O function and whose algorithm requires that iterations execute in a given order is a good candidate for the `sequential_loop` pragma.

Example 3–4: Using the #pragma sequential_loop Directive

```
main()
{
printf("This program counts from 1 to 100:\n");
#pragma sequential_loop
for (i = 0; i <= 100; i++)
    printf("%-d\n", i);
    .
    .
    .
}
```

3.8 Memory-Management Functions

There is an additional VAX C Run-Time Library (RTL) that you can use if you wish to use memory-management functions in programs that run in parallel. The VAX C RTL contains versions of the following functions that allow memory access between subprocesses during parallel processing:

- **malloc**
- **calloc**
- **free**
- **cfree**
- **realloc**

To use the new versions of these routines, you perform the following tasks:

1. Write the main function of a parallel program in VAX C.
2. Declare the memory-management functions by including *stddef.h* in your program.
3. Compile your program using the */PARALLEL* qualifier.
4. Link against the object library VAXCPAR.OLB.

The main function of your program must be named *main* or it must be declared using the *main_program* option. Otherwise, memory allocated by **malloc** cannot be accessed across subprocesses.

To avoid errors, the parallel-processing versions of the functions have different names from those of the normal memory-management functions. The *stddef.h* file contains macro definitions that make the correct versions of the functions available, depending on the current value of the predefined macro *CC\$parallel*; VAX C defines this macro according to the presence or

absence of the `/PARALLEL` qualifier on the command line (see Section 11.1.2 for more information). If you do not use `stddef.h` to declare the memory-management functions, a potential mismatch of versions may occur, causing unpredictable results in your program. See Section 10.4 for more information on including files into your program.

If you want to use the parallel-processing versions of the memory-management functions, you must follow the VAX C procedures for linking against object module libraries or for linking against shareable images. (See Section 1.4.5.2 for more information about linking against object module libraries; see Section 1.4.5.3 for more information about linking against shareable images).

The effect of using these versions of the functions is slightly different from that of using the normal versions. The **free** function does not return a value, and the **malloc** function does not set *errno*.

See the *VAX C Run-Time Library Reference Manual* for more information on these memory-allocation functions.

3.9 Tuning Issues Related to Parallel Processing

Parallel-processing programs may fail to execute because of insufficient system resources. You may have to adjust some resource-utilization parameters both for the entire system and for individual user accounts. You may also want to adjust some parameters to achieve better performance for programs executing in parallel. These considerations are addressed in the following sections.

You may also find it advisable to adjust system resources to accommodate the needs of the multiprocessing configuration of the VMS Debugger.

3.9.1 Customizing the Parallel-Processing Run-Time Environment

To tune the parallel-processing run-time environment in which a program is executed, you can use the logical names shown in Table 3-3.

Table 3–3: Logical Names Used for Run-Time Tuning

Logical Name	Use
FOR\$PROCESSES	Controls the number of processes used to execute a program in parallel (32 maximum)
FOR\$SPIN_WAIT FOR\$STALL_WAIT	Control CPU usage when waiting, for work or synchronization, in a program executing in parallel

You can define your own values for the logical names using the DCL commands `DEFINE` or `ASSIGN`. For example:

```
$ DEFINE FOR$PROCESSES 4
```

The values defined when a program starts parallel execution remain in effect until execution is completed.

The following sections describe the logical names, as follows:

- FOR\$PROCESSES (Section 3.9.1.1)
- FOR\$SPIN_WAIT (Section 3.9.1.2)
- FOR\$STALL_WAIT (Section 3.9.1.3)

3.9.1.1 Controlling the Number of Processes (FOR\$PROCESSES)

The logical name `FOR$PROCESSES` defines the number of processes to be used when executing a program in parallel. To define `FOR$PROCESSES`, you must specify a nonzero, positive number. The maximum number is 32. If you do not define a value for `FOR$PROCESSES`, a default value equal to the number of processors that are currently active on the system is used.

Being able to adjust the number of processes can be helpful for a variety of reasons, as follows:

- It enables you to execute your parallel program in a single process. This allows you to debug the logic within your parallel **for** or **while** loops as they execute in a serial, nonparallel fashion. (Note that running a program with parallel loops serially and in one process does not reduce the initialization overhead associated with the parallel loops.)
- It enables you to compare the performance impact of executing a parallel program with a varying number of processes.
- It enables you to gauge the tradeoffs between increasing system overhead and increasing execution time. For example, in a time-sharing environment, you may find it advisable to reduce the number of processes in order to minimize contention for system resources.

- It is useful when you are executing a program in parallel on a multiprocessor with more than two processors and you do not want to contend for the use of all the available processors.

3.9.1.2 Controlling Internal Spin Waits (FOR\$SPIN_WAIT)

The logical name FOR\$SPIN_WAIT allows you to tune the synchronization method used when a program runs in parallel.

The synchronization methods used by a program running in parallel in a multitasking environment must deal with two conflicting goals, as follows:

- To respond as quickly as possible to a synchronization flag. The common way to accomplish this is to repeatedly test for an appropriate flag in a shared storage location. Doing this test in a tight loop ensures a quick response when the flag is reset. This solution, however, conflicts with the second goal.
- To avoid wasting valuable CPU cycles that might be used by another program.

Because of this conflict, a tradeoff must be made between the fastest response to the synchronization flags and fairness to other programs.

VAX C allows you to affect this tradeoff by defining a value for FOR\$SPIN_WAIT. You can define it to be any nonnegative integer. This value specifies how many iterations of the spin-wait loop execute before the executing process gives up the processor and allows the VMS system to schedule another process.

Defined values for FOR\$SPIN_WAIT can be the following:

- A value of 0 is a special case that tells the run-time support to use the fastest synchronization at the expense of wasted CPU cycles. This value is appropriate for running a program in parallel on a system that is dedicated to running that single program.
- Other positive values tell the run-time support to use more or fewer spin-wait iterations, with higher values indicating more iterations. A value of 1 ensures the least wasted cycles—at the cost of the slowest synchronization response.

It is usually not necessary to define this logical name. The default value (1000) established by the run-time system should be adequate for most programs.

3.9.1.3 Controlling the State of a Process (FOR\$STALL_WAIT)

When a subprocess is waiting to work on a parallel loop, it can be either in an active state on the system or in an inactive state. When a subprocess is inactive, it becomes less responsive because it has to become active again before it can respond to the parallel loop.

As a second level of control over the internal spin waits in the parallel-processing environment, the logical name FOR\$STALL_WAIT allows you to control the time that a subprocess stays active on the system. To control how long it remains active, you define a value for the logical name FOR\$STALL_WAIT. This nonnegative value specifies the number of times that the subprocess will give up the CPU before becoming inactive.

Defined values for FOR\$STALL_WAIT can be the following:

- A value of 0 tells the run-time support to maintain the subprocess as active, so that the subprocess is more responsive when a parallel loop becomes available. A value of 0 is appropriate for programs that contain mostly parallel loops.
- Other positive values tell the run-time support to stay active for a longer or shorter interval, with higher values directing it to stay active longer. A value of 1 ensures that a subprocess waiting for a parallel loop will stay active for the shortest time interval. A value of 1 is appropriate when the program has large segments of code before, after, or between parallel loops.

It is usually not necessary to define this logical name. The default value (10 times the number of subprocesses) established by the run-time system should be adequate for most programs.

3.9.2 System Parameters Set with the SYSGEN Utility

When a parallel application is executed, much of the local memory and many external variables of the application are mapped to global sections (the VMS operating system's way of sharing data between processes). You must ensure that the number of global sections, global pages, and global page file sections required by a parallel application are available. To allow enough space for this global data, some of the system's sysgen parameters may need to be increased.

The most important sysgen parameters are GBLPAGFIL, GBLPAGES, and GBLSECTIONS. These parameters are not dynamic; your system must be restarted for any modifications to them to take effect. Adjust the parameters one at a time to avoid modifying some of them unnecessarily. Table 3–4 lists suggested values for the three sysgen parameters.

Your system manager should use the SYS\$UPDATE:AUTOGEN.COM command procedure to modify these sysgen parameters. Using AUTOGEN.COM, parameters related to those you are modifying are automatically changed for you. For details on how to use this procedure, see the installation guide for the operating system software installed on your system.

Table 3–4: Sysgen Parameters Requiring Changes for Parallel Processing

Parameter Name	Suggested Value ¹	Default	Minimum	Maximum	Unit	Dynamic
GBLSECTIONS	512	128	20	4095	Sections	No
GBLPAGES	32768	4096	512	-1	Pages	No
GBLPAGFIL	7000	1024	128	-1	Pages	No

¹The values listed under this heading are typical values.

The following sections provide more detail about the following parameters:

- GBLSECTIONS (Section 3.9.2.1)
- GBLPAGES (Section 3.9.2.2)
- GBLPAGFIL (Section 3.9.2.3)

3.9.2.1 Global Section Descriptor Count (GBLSECTIONS)

The GBLSECTIONS parameter sets the number of global section descriptors established in permanent resident memory at startup time. Each global section must have a descriptor. The number of global section descriptors determines the maximum number of global sections that can exist on the system at one time.

Each descriptor requires 32 bytes of permanent resident memory. To avoid wasting permanent resident memory, try to minimize the value you give to the GBLSECTIONS parameter.

If the count is not high enough, a diagnostic message is issued.

3.9.2.2 Global Page Table Entry Count (GBLPAGES)

The GBLPAGES parameter establishes the size of the global page table and the maximum number of global pages that can be created. For every 128 entries in the global page table, 4 bytes are added to permanent resident memory in the form of a system page table entry. (When you increase GBLPAGES beyond the default setting, you may want to increase the SYSMWCNT by 1 for each multiple of 128 entries that you add to the default setting.)

One way to calculate the number of global pages required to run an application using the VAX C parallel-processing support is to obtain a link map and add up the size of the psects that will be shared.

To get a link map, specify the /MAP/FULL qualifiers on your LINK command line. To calculate the approximate number of global pages required for your application, go through the link map and add up the decimal sizes of the psects for external variables and the \$LOCAL psect. (The link map gives you the size of the psects, in bytes.) In addition, the VAX C parallel-processing run-time support requires approximately 3 global pages for its own use; so add 1536 bytes to the number of bytes required for the psects. Then, to determine the number of global pages required for the application, divide the total number of bytes by 512.

If the count is not high enough, a diagnostic message is issued.

The GBLPAGFIL and GBLPAGES parameters must both be at least as large as the number of global pages required for your application.

3.9.2.3 Global Page File Limit (GBLPAGFIL)

The GBLPAGFIL parameter establishes the maximum number of global pages with page file backing store that can be created. Space for global page file sections is allocated from the paging file at startup time. When you increase this parameter you may want to increase the size of the paging files as well. You can check the current size of the paging files by using the DCL command SHOW MEMORY. For example:

```
$ SHOW MEMORY

[ other memory information removed ]

Paging File Usage (pages):
```

	Free	Reservable	Total
DISK\$PAGE: [PAGE]SWAFFILE2.SYS;1	68280	68280	79992
DISK\$PAGE: [PAGE]PAGEFILE2.SYS;1	73490	60190	79992

If the limit is not high enough, a diagnostic message is issued.

3.9.3 User Parameters Set with the AUTHORIZE Utility

You may need to adjust the PRCLM and PGFLQUO authorization quotas for any account that runs parallel applications. Adjust them using the following guidelines:

- The PRCLM quota determines the number of subprocesses that your process can create. For applications involving parallel **for** loops, it must be at least equal to the number you specify for the FOR\$PROCESSES logical name. (During debugging operations, one additional process must be available for the debugger.)
- The PGFLQUO quota is a pooled quota. It restricts the total pages that your processes can use in the system paging file. It is shared by all processes in a job so it may require an adjustment to allow for the additional processes used in parallel processing. It may need to be as high as the value that results from multiplying the total number of writable pages (shown in the Image Section Synopsis in the image map produced by the linker) times the number of processes.

If either of these quotas is not high enough, a diagnostic message is issued.

These quotas are adjusted using the Authorize Utility and are established only at login time. This implies that any current user of the account must log off and log back on before the quotas change for that user. The following user listing shows example settings for the PRCLM and PGFLQUO quotas:

```
Username: USER_J                               Owner: Joe User
Account: NONE                                   UIC: [360,100] ([USER_J])
CLI: DCL                                       Tables: DCLTABLES
Default: USRD$:[USER_J]
.
. [ other user information removed from this listing display ]
.
Prclm:          10  DIOLm:          18  WSdef:          300
Prio:           4   ASTlm:          30  WSquo:          500
Queprio:        0   TQElm:          20  WSextent:       2048
CPU:            (none)  Enqlm:        200  Pgflquo:       20000
.
. [ other user information removed from this listing display ]
.
```

Use the Authorize Utility's MODIFY command to change these quotas. For example:

```
UAF> MODIFY USER_J/PGFLQUOTA=23921
```

3.9.4 Other Tuning Considerations

Parallel-processing applications typically use large amounts of memory. To get better performance for an application, you may want to make adjustments to the working set size parameters (WSMAX, WSQUOTA, and WSEXTENT) both for the system and for user accounts. See the *Guide to VMS Performance Management* for information on how to adjust working set size.

VAX C Programming Concepts





This chapter is a VAX C tutorial for the experienced programmer. The topics covered in this chapter are as follows:

- C language overview (Section 4.1)
- VAX C language overview (Section 4.2)
- Writing a program (Section 4.3)
- Producing input and output (I/O) (Section 4.4)
- Conditional execution of code (Section 4.5)
- Values, addresses, and pointers (Section 4.6)
- Aggregate data structures (Section 4.7)

The text provides detailed examples and short tutorials, as well as pointers to other chapters in this guide. If you need detailed language information, see the more detailed chapters in this part of the guide.

4.1 C Programming Language Overview

The C language is a general-purpose programming language that is manageable due to its small size, flexible due to its ample supply of operators, and powerful due to its utilization of modern control flow and data structures. The C language was originally designed and implemented on a UNIX® system with the PDP-11. The designers of the language spoke of its functionality as follows:

“The [C] language . . . is not tied to any one operating system or machine; and although it has been called a ‘system programming

® UNIX is a registered trademark of American Telephone and Telegraph Company in the U.S. and other countries.

language' because it is useful for writing operating systems, it has been used equally well to write major numerical, text-processing, and database programs."¹

Like assembly language, C was not designed to accommodate the needs of any particular application. C manipulates and stores data with regard to the similarities of modern machine architecture. Despite their similarities, C is not as complex as assembler language and is not machine dependent. C is highly portable, which means that you can compile and run C source programs using different compilers on different machines.

There is no ANSI or other industry-wide standard for the C programming language at the time of publication (although the ANSI C committee is in the late stages of developing such a standard). Also, there is a consistency of functionality between implementations. There must be consistency if C is to be portable across systems, and this is one of the most desirable features of the language. So, not only must C source programs be portable, the language features themselves must produce the same effects on all systems when you compile and run programs.

The C language was developed in a UNIX system environment, and eventually was used to rewrite most of that operating system, so many standard methods of operation in C are related to UNIX. For instance, UNIX systems access files by a numeric file descriptor, so C implementations should provide functions to access files by file descriptor. In a UNIX system environment, you can expect a concise command structure, an ability to redirect output from one program or command to the input of another program or command, an ability to create asynchronous and synchronous subprocesses, and an ability to manipulate the operating system features without many restrictions and system safeguards.

Some standard C constructs include preprocessor directives and a run-time library of functions and macros. In a UNIX system environment, a preprocessor completes the tasks designated in the preprocessor directives located in the source code before the compiler takes any action.

Since the C language has no means to input and output information, a run-time library usually provides this service. If a run-time function produces side effects other than those produced in the UNIX system environment, the function's portability is questionable. For a complete discussion of portability, see the *VAX C Run-Time Library Reference Manual*.

¹ Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Englewood Cliffs, New Jersey: Prentice Hall, 1978), p. 1.

4.2 VAX C Programming Language Overview

The VAX C programming language incorporates the features that are fundamental to the C language and that exist in most C compilers. However, VAX C also provides features, unique to VAX C, that work directly and efficiently with the VMS operating system environment. You must decide which set of features of VAX C are most important to your programming needs: portability across systems or efficient use of the VMS operating system features. Choosing one set of features over the other has its benefits as well as disadvantages.

If you choose to program in VAX C so that your source programs are highly portable across systems, you sacrifice efficiency to some degree. For the VAX C Run-Time Library (RTL) to emulate UNIX system features, which it must do to maintain a satisfactory degree of portability, VMS features may have to be manipulated, causing a loss of efficiency. For example, a UNIX system accesses a file using a structure called a file descriptor, but VAX Record Management Services (RMS) access files using a variety of control structures. In VAX C, I/O functions appear to access files in the same manner as UNIX systems, but the run-time library actually manipulates RMS structures, making it appear as though you are working in a UNIX system environment. Most of the VMS manipulation is transparent to you, but can slow the execution of a program in some instances.

Most of the differences between VAX C and other implementations—differences that hinder the portability of source code—evolved due to the differences between VMS and UNIX systems. For example, it is difficult for VAX C to create an environment that gives you a great amount of control, as in a UNIX system environment, when a VMS system environment will not grant you such control; I/O redirection is not a part of the VMS command line syntax; creating subprocesses on VMS systems is not as efficient as it is on UNIX systems; and, VMS high-level languages do not implement preprocessors in the same manner as languages on a UNIX system. In this guide, differences between VAX C and other implementations are flagged in the text so that you can use nonportable VAX C constructs efficiently.

If you choose to program in VAX C so that your program works with the VMS system in an efficient manner, you sacrifice, to some degree, the option of being able to port your programs to and from other systems. For example, you can call the VMS Run-Time Library (RTL) routines within VAX C programs.

However, you can have portability and efficiently access the powerful VMS environment. You can use special constructs of VAX C and the DIGITAL Command Language (DCL) (such as the VAX C preprocessor substitutions and the DCL command-line qualifier `/STANDARD=PORTABLE`). These constructs allow you to execute some segments of code only when running on a VMS system, and to execute other segments of code when running on systems other than a VMS system. For more information about the preprocessor directives, see Section 11.1.7. For information about the VAX C compilation qualifiers, see Section 1.3.2.

4.3 Writing a Program

In an effort to keep the examples in this tutorial simple so that you can concentrate on the concepts, the first program presented here adds two numbers and stores the total in a variable. Example 4-1 shows how to code such a program.

Example 4-1: Simple Addition in VAX C

```
❶ /* This program adds two numbers and places the sum in      *
   * the variable total.                                       */

❷ main()                                                       /* The function name "main" */
  {                                                            /* Begins function body   */
❸     int total;                                             /* Variable of type "int"  */
                                                                /* Blank lines are allowed */
❹     total = 2 + 2;                                         /* Answer placed in "total" */
  }                                                            /* Ends the function body  */
```

Key to Example 4-1:

- ❶ The text bordered by the characters `(/*)` and `(*/)` are comments. You cannot place comments within comments (that is, they cannot be nested), but you can place comments anywhere white space is allowed. *White space* is an area within the source code where blank spaces or blank lines separate code. In later chapters, permitted white space is defined for VAX C constructs.
- ❷ VAX C programs are comprised of user-defined external functions that cannot be nested. Here, a function named `main` is defined. In VAX C, execution of a program begins at either a function named `main` or at a function defined using the `main_program` option, or both. If a user-specified `main` function does not exist, the first function in the program stream at the time external references are resolved is the default `main`.

function. The `main_program` option is VAX C specific and is not portable. For more information about the syntax and usage of the `main_program` option, see Section 5.1.1.

VAX C functions have methods of exchanging information using parameters and arguments. In the function definition of `main`, the lack of parameters is designated by the empty parentheses. In Example 4-1, the function `main` cannot receive information using parameters.

To specify parameters in a function definition, list the parameter identifiers within the parentheses and separate them with a comma (,). You must declare the parameters before the beginning of the body of the function. If you call a function from within function `main` (you normally do not call the `main` function from another part of your program), the function name is followed by a list of arguments delimited by parentheses and separated by commas. The number of arguments must correspond with the number of parameters in the function declaration. In Example 4-1, there are no function calls.

The function performs its task as determined by the statements found in the body, and may or may not return a value to the calling expression. The body of the function `main` is delimited by braces ({}). They are similar to the **DO-END** of PL/I, or the **BEGIN-END** of Pascal. The body usually contains one or more **return** statements. A **return** statement specifies what, if anything, is returned to the expression that called the function. Depending on the set-up of the function, you can omit the **return** statement, and its return value will remain undefined. If a function does not return a value, you can declare the function to be of data type **void**. For more information about functions, see Section 5.1.1. For more information about function parameters, see Section 5.1.2.

- ③ In the example, the variable `total` is declared and defined within the function `main`. You usually declare all variables before referencing them within the program. Declarations end with a semicolon (;). If you declare a variable, you specify its data type. Data types specify the amount of storage required and how to interpret the stored object. For example, variable `total` is of the data type **int** (integer), the object of which requires 32 bits (4 bytes or 1 longword) of memory. VAX C interprets variables of type **int** as integers having a positive or negative sign (or zero).

When you define a variable, you specify its storage class, which affects its location, lifetime, and scope. Variables declared within a function have a default storage class of **auto** (automatic). Variables of this storage class receive storage space when the function is activated and storage is freed when control of the calling function resumes. Not all storage classes are implemented by default. You can specify all VAX C

storage classes and may place the storage-class keyword either before or after the data-type keyword in the variable declaration.

Data types and storage classes are very important when determining the scope of a variable. For more information about data types, see Chapter 8. For more information about storage classes, see Chapter 9.

Keywords are the reserved words used to identify data types (such as **int**, **double**), storage classes (such as **auto**, **globalvalue**), statements (such as **if**, **goto**), and operators (such as **sizeof**). Keywords are predefined and cannot be redeclared. You cannot use these words to identify variables and functions in your programs. You must express keywords in lowercase letters. For a list of the VAX C keywords, see Section 5.6.

VAX C is a case-sensitive language. You can declare variables such as `total` in any mixture of upper- or lowercase letters. If you reference variable `total` in your program, the reference also must be lowercase. For example, if you attempt to reference variable `Total`, an error occurs; the compiler does not recognize the variable name due to the initial capital letter.

- ④ The sum of $2 + 2$ is stored in variable `total`. This is accomplished using a valid VAX C statement. You can use any valid expression as a statement by ending it with a semicolon (;). Identifier `total` is a declared variable; the equal sign (=) and the plus sign (+) are valid VAX C operators; and the numbers being added are valid constants. For more information about the various VAX C statements, see Chapter 6. For more information about the VAX C operators, see Chapter 7.

4.4 Producing Input/Output (I/O)

The C language includes no facilities to administer input or output (I/O). However, all implementations must have methods that allow the programs and users to communicate. The lack of communication in Example 4-1 is inconvenient; there is no way to know if the program assigns the correct value of 4 to variable `total`. You can use a VAX C Run-Time Library (RTL) function to output the value of variable `total` to the terminal.

All C compilers are accompanied by a run-time library of functions and macros in order to perform input, output, and various tasks related to specific operating environments. The VAX C RTL provides many of the functions and macros that are included with other implementations of the C language. In addition, there are functions that work directly and efficiently with the VMS environment.

Before you can execute any of the example programs in this manual, you have one of two options. You can link against the VAX C RTL in an object code library or in a shareable image. Both methods require instructions to be passed to the linker so that the linker knows the location of the correct versions of the functions or macros you wish to use.

If you want to use the VAX C RTL as an object library, you *must* define, in the correct order, the libraries the linker must search to resolve references to VAX C RTL functions. All the VAX C RTL object code modules are located in the libraries SYS\$LIBRARY:VAXCCURSE.OLB, SYS\$LIBRARY:VAXCRTL.G.OLB, SYS\$LIBRARY:VAXCRTL.OLB. and SYS\$LIBRARY:VAXCPAR.OLB. To determine in which order to define these libraries, see Section 1.4.5.2. For general information about libraries, see Section 1.4.5.1.

If you prefer to use the VAX C RTL as a shareable image, see Section 1.4.5.3 for more information. VAX C RTL macro references within program source code look just like function references. However, the compiler replaces macro references with VAX C source code at an early stage in the execution process. The compiler locates VAX C RTL macro source code in the .H definition files provided with VAX C. If your system manager extracted these .H files during installation, you can access the files in the directory SYS\$LIBRARY. For example, you can type the STDIO.H file at your terminal with the following command:

```
$ TYPE SYS$LIBRARY:STDIO.H 
```

If this command causes an error, see your system manager about the extraction of the .H files during installation. It is a good idea to type or print all of the .H files to see the macros and definitions provided with VAX C.

You can also locate the .H definition files in text library VAXCDEF.TLB located in directory SYS\$LIBRARY. This guide refers to the .H files as definition modules since they can be accessed as modules in this text library.

For more information about macros, see Section 10.1.3. For more information on the various methods of accessing VAX C RTL functions, see the *VAX C Run-Time Library Reference Manual*.

Example 4–2 shows that by using the VAX C RTL function **printf**, a VAX C program can print a message to the terminal.

Example 4-2: Output of Information

```
/* This program adds two numbers, assigns the value 4 to *
 * variable total, and then prints the answer on the *
 * terminal screen. */
❶ #include <stdio.h> /* Good programming practice when *
 * using I/O functions. */
main()
{
    int total;
    total = 2 + 2;
    /* Print intro string */
❷ printf("Here is the answer: ");
    printf("%-d.", total); /* Print the answer */
}
```

Key to Example 4-2:

- ❶ When you are using any of the I/O functions, it is good programming practice to include the definition module that is appropriate for that function (see the *VAX C Run-Time Library Reference Manual*). In the case of **printf**, you should include the *stdio* module, which is located in the text library `SYS$LIBRARY:VAXCDEF.TLB`. This module contains function prototypes and macro definitions that are used by many I/O functions.
- ❷ The VAX C RTL function **printf** writes to the standard output file (the terminal screen). The first call to the VAX C RTL function **printf** passes a string as the argument. The second call to **printf** passes a string with special formatting characters and a variable as arguments. Within the formatting string, the percentage sign (%) is replaced by the value of `total`, the minus sign (-) left-justifies the output, and the letter `d` forces the value of the argument to be expressed as a decimal number. The period (.) prints immediately after the value of `total`.

The output for Example 4-2 is as follows:

```
Here is the answer: 4.
```

If you want to print the value of `total` on a separate line, then the newline character (`\n`) must be added to the string. Example 4-3 shows how to output on two lines.

Example 4-3: Output Using the Newline Character

```
/* This program adds two numbers, stores the sum in the      *
 * variable total, and then prints the answer on two        *
 * separate lines on the terminal screen.                    */

#include stdio

main()
{
    int total;
    total = 2 + 2;

    printf("Here is the answer . . . \n");                /* Print intro string */
    printf("%-d.", total);                                /* Print the answer */
}
```

Output from this program is as follows:

```
Here is the answer . . .
4.
```

Now that a program producing output has been presented, it is necessary to compile, link, and execute the program using DCL to see the results. Compiling a program translates the source code to object code; linking a program organizes storage and resolves external references (for example, references to VAX C RTL functions); and running a program executes the image.

In the VMS environment, a file is distinguished by a file name and a file extension. Choose the file name so that the file is easily identifiable to the user. Choose the file extension to reflect the functionality of the file. For example, the file name ADDITION.C is a good name for a VAX C source program. The file extension .C is the default file extension for the VAX C compiler. If the file name ADDITION is given to the VAX C compiler, the compiler will look for the file ADDITION.C.

After you create and name your program, the program can be compiled, linked, and executed as follows:

```
$ DEFINE LNK$LIBRARY SYS$LIBRARY:VAXCTRL.OLB RETURN
$ CC ADDITION.C RETURN
$ LINK ADDITION.OBJ RETURN
$ RUN ADDITION.EXE RETURN
Here is the answer . . .
4.
$
```

The .OBJ and .EXE extensions are the default file extensions assigned to the object file and the image file, respectively.

You may have to define more libraries to the linker or use shareable images in order to use VAX C RTL functions in your program. The definition in Example 4–3 is sufficient to execute all example programs in this chapter, if you have the object libraries installed on your system. After you define the libraries, you do not have to define them again for the remainder of the terminal session (until you log out). For more information on the compilation process, see Section 1.3. For more information on accessing the VAX C RTL in an object library, see Section 1.4.5.2. For information on using the VAX C RTL as a shareable image, see Section 1.4.5.3.

4.5 Conditional Execution of Code

There will be occasions when you must execute one or more VAX C statements given a certain condition. There will be other occasions when you must execute one or more VAX C statements repeatedly, within the body of a loop, until you meet a certain condition. There are several statements in VAX C that accomplish these tasks. These statements are the **if** statement, the **switch** statement, the **do** statement, and the **for** statement. For information about the **while** statement, another statement that loops until meeting a condition, see Section 6.4.2.

4.5.1 The if Statement

When executing one or more VAX C statements given a certain condition, you can use the **if** statement. Example 4–4 shows a program using the **if** statement.

Example 4-4: Conditional Execution Using the if Statement

```
/* This program asks the user to guess a letter. The      *
 * program tells whether the answer's correct or         *
 * incorrect. The program is hard coded to accept 'a' or *
 * 'A' as the correct letter.                            */

#include stdio

main()
{
    char ch;                /* Declare a character      */
                           /* Ask the user to guess */
    printf("Guess which letter I'm thinking of!\n");

    ❶ ch = getchar();        /* Get the character      */
                           /* Correct = "a" or "A" */
    ❷ if (ch == 'a' || ch == 'A')
        printf("You're right!");
    else
        /* If incorrect guess */
        {
            printf("You're wrong.\n");
            printf("You'll have to try again!");
        }
}
```

Key to Example 4-4:

- ❶ The VAX C RTL function **getchar** retrieves a character from the standard output device (the terminal). The program pauses, waiting for the user to type a character and to press the RETURN key. The function **getchar** retrieves one character and ignores any others that are typed in.
- ❷ If the letter that the user types is either 'a' or 'A', then a message stating that the choice is correct is displayed. If any other letter is typed, then a message stating that the choice is incorrect prints. The equality operator (==) compares the variable ch with the constants 'a' and 'A'. The logical OR operator (||) presents the condition to test. If there is more than one statement to be executed upon condition, then you must enclose the statements within braces ({}). A statement or statements enclosed within braces is called a *block* or *compound statement*. The concept of blocks is important when determining the scope of variables. For more information about blocks, see Section 5.7.

The output for Example 4-4 is as follows:

```
$ RUN EXAMPLE4 [RETURN]
Guess which letter I'm thinking of!
B [RETURN]
You're wrong.
You'll have to try again!
```

4.5.2 The switch Statement

The **switch** statement can perform the same task as the **if** statement does in Example 4-4, but **switch** is useful when many conditions must be tested. Example 4-5 uses the **switch** statement.

Example 4-5: Conditional Execution Using the switch Statement

```
/* This program plays the same guessing game as the      *
 * previous example except that it uses the switch      *
 * statement.                                           */
#include stdio
❶ #include ctype           /* Include proper module    */
main()
{
    char ch;

    printf("Guess what letter I'm thinking of!\n");
    ch = getchar();
    ❷ ch = _tolower(ch); /* Convert "ch": lowercase */
    switch(ch)          /* Examine "ch"           */
    {                  /* Body of switch statement */
        case 'a' :
            printf("You're right!");
            return;

        default : /* Any other answer           */
            printf("You're wrong.\n");
            printf("You'll have to try again!");
    }
}
```

Key to Example 4-5:

- ❶ When using the macro **_tolower**, you must include the definition module *ctype* in the compilation process. The module *ctype* is located in the text library `SY$LIBRARY:VAXCDEF.TLB`, and defines macros and constructs used for character processing and classification.

In VAX C, the preprocessor directives are processed by an early phase of the compiler, not by a separate program as the name preprocessor implies. Directives, unlike other VAX C lines of source code, begin with a pound sign (#). The pound sign must appear in column 1—the far left margin of your source file. Do not end preprocessor directives with a semicolon.

The module *ctype* is not the only module that contains macros and definitions used by the VAX C RTL functions; there are several ways to include definitions in the program stream. For more information about the VAX C RTL and the definition modules, see the *VAX C Run-Time Library Reference Manual*.

- ② The compiler replaces the reference to the **_tolower** macro with a line of VAX C source code that, when the program is run, translates the value of the variable `ch` to a lowercase letter. To see the macro definition of **_tolower**, print the file `SYS$LIBRARY:CTYPE.H` if it is available on your system. For more information about the possible side effects of macros, see Section 10.1.3.

The output for Example 4–5 is as follows:

```
$ RUN EXAMPLE5 RETURN
Guess which letter I'm thinking of!
A RETURN
You're right!
```

The **switch** statement executes one or more of a series of cases based on the value of the expression in parentheses. If the value of variable `ch` is 'a', then the statements following the label case 'a' : are executed. In Example 4–5, the **_tolower** macro translates all alphabetic answers to lowercase letters, so there is no need to test for uppercase letter 'A'.

When a case label is matched with the value of expression `ch`, all the statements following the remaining case labels are executed until the compiler encounters a **break** statement (which terminates the immediately enclosing statement), a **return** statement (which terminates the enclosing function), or the end of the **switch** statement. The statements following the **default** label are executed if the value of the expression does not match any of the other case labels. For more information about **switch** statements, see Section 6.3.2.

4.5.3 Loops

In the previous examples, you could only guess once during the execution of the program. To guess another letter, you had to execute the program again. If you want to execute a segment of code repeatedly until a condition is met, you may use a loop. Some loops execute a block of statements, known as the *loop body*, a specified number of times. Some loops test for a condition first and then execute the body of the loop if the condition is true. Some loops execute the loop body and then test for a condition, which guarantees at least one execution of the body. In VAX C, this last loop is called the **do** statement. Example 4-6 shows that you can use the **do** statement to alter the letter-guessing program.

Example 4-6: Looping Using the do Statement

```
/* This program plays the same guessing game as the      *
 * other examples except that the user must guess until *
 * the answer is correct. This is accomplished using a  *
 * do statement.                                       */

#include stdio
#include ctype

main()
{
    char ch;

    printf("Guess what letter I'm thinking of!\n");
    printf("Keep guessing till you get it!\n");

    do
        {
            ch = getchar();
            ch = _tolower(ch);
            switch(ch)
            {
                case 'a' :
                    printf("You're right!");
                    return;

                /* Ignore RETURN (newline) ch */
                case '\n':
                    break;
            }
        }
}
```

(continued on next page)

Example 4–6 (Cont.): Looping Using the do Statement

```
                default :
                    printf("You're wrong.\n");
                    printf("You'll have to try again!\n");
                }
                /* End of switch statement */
            }
            /* End of do loop body */
            /* Condition to be tested */
2 while(ch != 'a');
}
```

Key to Example 4–6:

- 1 In this example, the case label tests to see if the value of the character is a newline character (`\n`). The newline character is entered when you press the RETURN key. If it is the newline character, the character is ignored and a new character is taken from the terminal.
- 2 The **while** expression at the end of the **do** statement uses the not equal to operator (`!=`) and translates as follows: “while the variable `ch` is not equal to `'a'` AND `ch` is not equal to `'A'`.”

The output for Example 4–6 is as follows:

```
$ RUN EXAMPLE6 [RETURN]
Guess which letter I'm thinking of!
Keep guessing till you get it!
B [RETURN]
You're wrong.
You'll have to try again!
A [RETURN]
You're right!
```

You can use the **for** statement to specify the number of times to execute the loop body; in regard to the previous examples, it can be used to limit the number of guesses that the user may attempt. Example 4–7 shows how to use the **for** statement.

Example 4-7: Looping Using the for Statement

```
/* This program plays the same guessing game as the      *
 * previous examples except that the user is limited to  *
 * three guesses. This is accomplished using a for      *
 * statement.                                           */

#include stdio
#include ctype

main()
{
    char    ch;
    int     i;                /* An incrementor for loop */

    printf("Guess what letter I'm thinking of!\n");
    printf("You have three guesses. Make them count!\n");
                                /* Do the following 3 times */
    ❶ for (i = 1; i <= 3; i++ )
        {                    /* Beginning of loop body */
            ch = getchar();
            ch = _tolower(ch);
            switch(ch)
            {
                case 'a' :
                    printf("You're right!");
                    return;
                case '\n':
                    ❷ --i;
                    break;

                default :
                    printf("You're wrong.\n");
                    if (i != 3)
                        printf("You'll have to try again!\n");
            }                /* End of switch statement */
        }                    /* End of for loop body */
    printf("Sorry, you ran out of guesses!");
}
```

Key to Example 4-7:

- ❶ In the example, the **for** statement controls how many times the body of the loop is executed. The first expression inside the parentheses following the keyword **for** initializes loop incrementor *i* to the value 1. The second expression establishes an upper bound; the value of variable *i* is not to exceed 3. The third expression establishes the increment or decrement value of the variable that will be executed after every execution of the loop body. The double plus signs (**++**) are the increment operator; they increase the value of a variable by the integer 1. The loop body is executed, and the value of variable *i* increases by 1 each time, until the value of *i* is greater than 3.

- ② The double minus signs (--) are the decrement operator. The decrement operator is used in this example to subtract 1 from the value of variable `i` so that newline characters are not counted as the guess of a letter.

A sample output for Example 4-7 is as follows:

```
$ RUN EXAMPLE7 [RETURN]
Guess which letter I'm thinking of!
You have three guesses. Make them count!
B [RETURN]
You're wrong.
You'll have to try again!
C [RETURN]
You're wrong.
You'll have to try again!
U [RETURN]
You're wrong.
Sorry, you ran out of guesses!
```

4.6 Values, Addresses, and Pointers

In VAX C, every variable has two types of values: a memory location and a stored object. In VAX C, an *lvalue* is the variable's address in memory, and an *rvalue* is the stored object. Consider the following example:

```
put_it_here = take_this_object;
```

This assignment statement is not very different from statements in other programming languages, but think about the differences between locations in memory and objects stored in memory. This assignment takes *take_this_object's* rvalue and places it in memory at *put_it_here's* lvalue.

Consider the following VAX C assignment statement:

```
int x = 2, y;
/* put_it_here = take_this_object; */
    y      =      x;
```

The two distinct variables have different memory locations (lvalues), but, after the assignment statement, they contain objects of the equivalent value 2.

A variable's rvalue can be an integer, a real number, a character string, or a data structure. The rvalue can also be the address of another variable. In other words, a variable's rvalue can be another variable's lvalue. In this case, one variable points to another variable.

A declaration of a variable whose rvalue is a pointer to another variable is as follows:

```
int *pointer;
```

The indirection operator (`*`) specifies that the variable is a pointer, which in this example points to an object of data type **int**. Pointers are declared as pointing to an object of a particular data type.

You can assign the address of a variable to the pointer as follows:

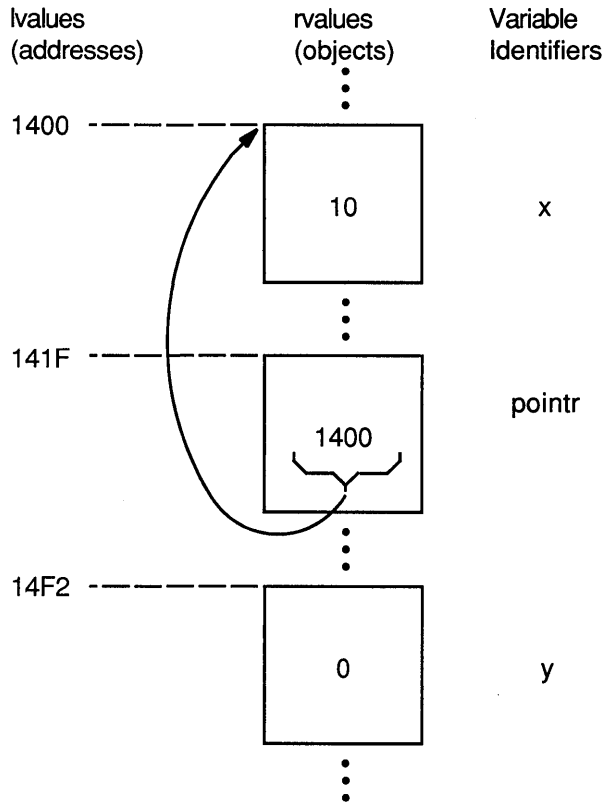
```
static int *pointer;           /* Declarations          */
static int x = 10, y = 0;
                               /* Assignment          */
    pointer = &x;
```

The rvalue of the variable `pointer` is the lvalue of variable `x`. In other example assignment statements, the rvalue of the variable on the right side of the equal sign (`=`) was taken. In this example, the ampersand (`&`), which is the address of operator, translates to the following: “take the lvalue of this variable instead of its rvalue.”

The **static** keyword specifies the **static** storage class. For general information about storage classes and the scope of variables, see Section 9.1. For information about the **static** storage class, see Section 9.4.

Figure 4-1 shows the difference between rvalues and lvalues.

Figure 4-1: rvalues, lvalues, and Assigning Pointers



ZK-3019-GE

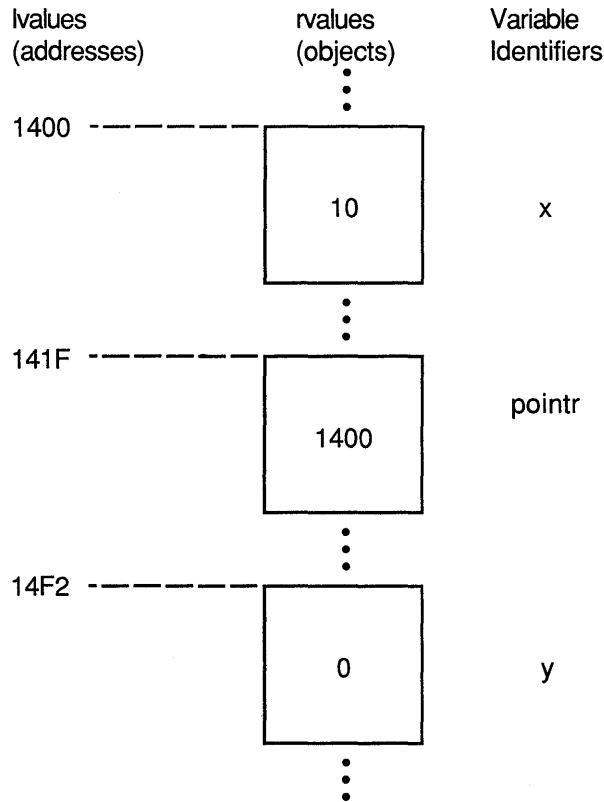
The value of the variable `pointer` contains the address of variable `x`. Remember that the location of variables in memory and the order in which the compiler processes them is unpredictable and left to the discretion of the compiler.

After you assign an address to the pointer, you will want to use it. For example, if you want to assign `x`'s rvalue to a variable `y`, you can use the pointer in a VAX C statement as follows:

```
y = *pointer;
```

The asterisk (*) is the VAX C indirection operator; the object of the variable being pointed to by `pintr` is assigned to `y`. The indirection operator translates as follows: "the rvalue of this variable points to some other variable, so go to that location and access the stored object." Figure 4-2 shows the status of the variables after you execute the last code example.

Figure 4-2: The Indirection Operator in Assignments



ZK-3020-GE

For detailed information about lvalues and rvalues, see Section 7.1. For more information about pointers, see Section 8.5.

4.7 Aggregates

The variables used in the previous examples were either pointers or single objects that could be manipulated, in their entirety, in an arithmetic expression. These types of variables are called *scalar* variables. The VAX C data structures—arrays, structures, and unions—are called *aggregates*. Aggregates are comprised of segments called *members*. Members are sections of the structure that you can declare to be of a scalar or an aggregate data type.

The following sections discuss arrays, character strings, structures, and unions.

4.7.1 Arrays and Character Strings

An *array* is a data structure whose members are of the same type. Members of arrays can be any of the scalar or aggregate data types.

In VAX C, character strings are represented internally as arrays of type **char**. You may declare and initialize a character string as a character-string variable using the indirection operator (*), as an array of a specified number of members, or as an array of an unspecified number of members, as follows:

```
char *str = "Hello";
char string[6] = "Hello";
char string[] = "Hello";
```

In VAX C, all character strings end with the NUL character ('`\0`'). In the previous example, the NUL character is appended to Hello making the string six characters in length. When assigning strings to character-string and array variables within the executable portion of the program, you must use the string-handling VAX C RTL functions. For more information about the string-handling functions, see the *VAX C Run-Time Library Reference Manual*. Example 4-8 shows the use of character strings and arrays.

Example 4–8: Character-String Constants and Arrays

```
/* This program plays the same guessing games as the      *
 * previous examples except that it uses character-      *
 * string constants and arrays.                          */
#include stdio

main()
{
    char ch;                               /* Declare a character */
                                           /* Initialize messages */
    char *greeting = "Guess which letter I'm thinking of!";
    char *message1 = "You're right!";
    char *message2 = "You're wrong.";
    char *message3 = "You'll have to try again!";
    char correct[2];
    correct[0] = 'a';                       /* Store correct letters */
    correct[1] = 'A';

    printf("%s\n", greeting);              /* %s = char string */
    ch = getchar();

    if (ch == correct[0] || ch == correct[1])
        printf("%s", message1);
    else
    {
        printf("%s\n", message2);
        printf("%s", message3);
    }
}
```

The output for Example 4–8 is as follows:

```
$ RUN EXAMPLE8 
Guess which letter I'm thinking of!
B 
You're wrong.
You'll have to try again!
```

For more information about arrays, see Section 8.7. For more information about character strings, see Section 8.8.

4.7.2 Structures and Unions

Structures and unions are aggregates whose members can be of different types. Structures and unions are declared using the keywords **struct** and **union**, an optional tag name, and a list of member declarations delimited by braces (`{ }`). A member of a structure or a union is a declared segment of the data structure. The syntax for declaring a member is the same as for declaring any variable. The structure or union tag is a name that can be used when declaring structure or union variables of the same type elsewhere

in the program. Members of structures and unions may be referenced as follows:

```
main()
{
    struct optional_tag          /* Tag = optional_tag */
    {
        char    letter_1;
        char    letter_2;
        int     number;
    } characters = {'a', 'b', 59}; /* Variable = characters */
    characters.letter_1 = characters.letter_2;
}
```

You may reference members using the structure or union variable name, directly followed by a period (.), directly followed by the member name. As in the previous example, structures are initialized using a variable name and an assignment operator (=) immediately following the declaration of the members. The values of the members are delimited by braces and separated by commas (,). The address of the first member of a structure begins, in memory, at the base of the data structure, which is referred to as offset zero.

Unions are declared in the same way as structures, but all members in a union begin at offset zero. Unlike structures, unions cannot be initialized. The size of the union in memory is as large as its largest member. When the single storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered. Example 4-9 shows the nature of unions.

Example 4–9: Single Storage Allocation of Unions

```
/* This example shows the storage maintenance of          *
 * unions with different size members.                    */
#include  stdio
main()
{
    union
    {
        char  lastname[8];    /* Array for a last name    */
        char  firstinit;     /* Char. for first initial */
    } overlap;
                                /* Copy and print members */
    strcpy(overlap.lastname, "Jackson");

    printf("%s\n", overlap.lastname);
    overlap.firstinit = 'M';
    printf("%c\n", overlap.firstinit);
    printf("%s\n", overlap.lastname);
}

```

The output for Example 4–9 is as follows:

```
$ RUN  EXAMPLE9.EXE  RETURN
Jackson
M
Mackson
```

The VAX C RTL function **strcpy** copies the second string argument into the first array argument. When assigning values to smaller union members, the compiler does not fill the remaining space with NUL characters (`'\0'`); whatever was in memory at the time remains. For more information about structures and unions, see Section 8.9.

Example 4–10 shows a structure definition and its usage.

Example 4-10: Structures

```
/* This program plays the same guessing game as the      *
 * previous examples except that it uses a structure.    */
#include <stdio>

main()
{
    char ch;
    char *greeting1 = "Guess which letter I'm thinking of!";
    char *greeting2 = "You've 3 guesses. Make them count!";
    char *message1 = "You're right!";
    char *message2 = "You're wrong.";
    char *message3 = "You'll have to try again!";
    char *message4 = "Sorry, you ran out of guesses!";
    int i;

    /* Store information */
    /* Structure tag = storage */
    ① struct storage
        {
            char small_a; /* One correct letter */
            char capital_a; /* Another correct letter */
            char newline_ch; /* newline character */
            int num_guesses; /* Number of guesses */
        };

        /* Declare "letter" */
        /* using tag "storage" */
    ② struct storage letter = {'a', 'A', '\n'};

    letter.num_guesses = 3;
    printf("%s\n", greeting1);
    printf("%s\n", greeting2);

    for (i = 1; i <= letter.num_guesses; i++)
    {
        ch = getchar();
        if (ch == letter.small_a || ch == letter.capital_a)
        {
            printf("%s", message1);
            return;
        }
        else
            if (ch == letter.newline_ch)
                --i;
            else
            {
                printf("%s\n", message2);
                if (i != 3)
                    printf("%s\n", message3);
            }
    }
}
```

(continued on next page)

Example 4-10 (Cont.): Structures

```
    }                               /* End of for loop body    */
    printf("%s", message4);
}
```

Key to Example 4-10:

- ① In the example, the structure declaration with the tag storage has four members. The first three members are of type **char**. The last member is of type **int**.
- ② The variable letter is declared using the tag storage and individual members of the structure are initialized. The equal sign initializes the members of the structure variable with constants. The constants are separated by a comma and are delimited by braces. The number of initializing constants cannot exceed the number of members. However, as in this example, you may omit constants; the compiler pads the uninitialized member (in the example, member num_guesses) with zeros. However, you cannot initialize a member in the middle of any aggregate without initializing the previous members.

The output for Example 4-10 is as follows:

```
$ RUN EXAMPLE10 [RETURN]
Guess which letter I'm thinking of!
You've 3 guesses. Make them count!
B [RETURN]
You're wrong.
You'll have to try again!
C [RETURN]
You're wrong.
You'll have to try again!
U [RETURN]
You're wrong.
Sorry, you ran out of guesses!
```

After executing these program examples, you are well on your way to programming in VAX C.

Program Structure

A VAX C program is a group of user-defined functions that cannot be nested (you cannot define functions within other function definitions). This chapter describes the following components of program structure:

- Function definitions (Section 5.1)
- Function declarations (Section 5.2)
- Using function prototypes (Section 5.3)
- Using function parameters and arguments (Section 5.4)
- Identifiers (Section 5.5)
- Keywords (Section 5.6)
- Blocks (Section 5.7)
- Comments (Section 5.8)
- LINT-like functionality (Section 5.9)

5.1 Function Definitions

You may declare or define functions you wish to call or use in a VAX C program. You may or may not have to declare user-defined functions before you call them. This depends on what type of value the function returns, and the position of the function definition within the program. The following sections explain the rules for defining functions.

In a function definition, you specify the VAX C statements that execute whenever you call the function. You also specify the parameters (if any) of the function. The parameters of a function provide a means to pass data to the function. See Section 5.4 for a detailed discussion about using parameters and arguments.

Example 5-1 presents an example of two function definitions.

Example 5-1: Case Conversion Program

```
/* This program converts its input to lowercase. The *
 * first function passes control to the second function *
 * to convert a letter. Comments are located to the *
 * right of the code. */

#include stdio /* To use I/O definitions */
main()
① {
    FILE *infile, *outfile; /* Declare files */
    int i, c, c_out; /* Open "infile" for input */
    infile = fopen("ex113.in", "r"); /* Open "outfile" for output */
    outfile = fopen("ex113.out", "w");
    /* While not end of file... */
    /* Get a char from the file */
    while ((c = getc(infile)) != EOF)
    {
        c_out = lower(c); /* Send char to "lower" */
        /* Output the char to file */
        putc(c_out, outfile);
    }
    return; /* Optional return statement */
}

/* ----- *
 * Beginning of the next function definition: *
 * ----- */

/* Function and parameter *
 * name */
② lower(c_up)
③ int c_up; /* Declare parameter type */
{ /* Beginning function body */
    /* If capital, convert */
    if (c_up >= 'A' && c_up <= 'Z')
        return c_up - 'A' + 'a';
    else /* Else, return as is */
        return c_up;
} /* End of function body */
/* End function definition */
```

Key to Example 5-1:

- ① Program execution begins with function main. A left brace ({) signifies the beginning of the function body; a right brace (}) signifies the end of the body. The function body is any set of valid VAX C statements or

declarations. Usually, the body includes one or more **return** statements, as shown here. A **return** statement can specify an expression whose value is returned to the calling function. If the expression is omitted, the returned value is undefined in the calling function. If the **return** statement is not included, the function terminates when the right brace is encountered, and its return value is undefined.

- ② The identifier `lower` begins a new function definition; function `lower` has the single parameter `c_up`. Although function `main` has no parameters, the parentheses must be present.
- ③ The next statement, `int c_up`, declares the parameter's data type; in this case, **int** (integer). The declaration is omitted if the function has no parameters; furthermore, declarations at this place in the program should specify only the names of parameters, not the names of other variables used in the function body. For more information about data types and declarations, see Section 8.2.

For more information about the VAX C operators used in the previous example, see Section 7.3.

5.1.1 Main Function and Function Identifiers

The execution of a program begins at the function whose identifier is `main`, or, if there is no function with this identifier, at the first function seen by the VMS Linker. In Example 5-1, the main function physically precedes the function `lower`, but the two function definitions can appear in the reverse order. The word `main` is not a language keyword, so it may be used for other purposes in the program.

Function names have compile-time scope rules that are different from those that apply to other identifiers. Any valid function identifier followed by a left parenthesis is declared implicitly as the name of a function whose storage class is external and whose return value is of the data type **int**. For more information about scope and storage classes, see Section 9.1.

Between the definition of a function's identifier and the declaration of its parameters, you can write the following option:

```
main_program
```

The `main_program` option identifies the function as the main function in the program. It is not a keyword, and it can be expressed in either upper- or lowercase. Use the `main_program` option when the program does not contain a function named `main` and when you do not want the program's execution to begin at the first function linked. For example, the following

definition establishes function lower as the main function; execution begins there, regardless of the order in which the function is linked:

```
char lower(c_up)
MAIN_PROGRAM
int c_up;
{
    .
    .
    .
}
```

NOTE

The `main_program` option is VAX C specific and is not portable.

5.1.2 Parameter List Declarations

Example 5-1 shows only one of two methods to declare function parameters. The first method is as follows:

```
lower( c_up )
int c_up;
{
    .
    .
    .
}
```

To make your code concise, you may list the data types of the function parameters within the parameter list. If you use this method, your function definition also serves as a function prototype. See Section 5.3 for more information about the effect of function prototypes.

The second method of declaring parameter data types is shown in the following code example:

```
lower( int c_up )
{
    .
    .
    .
}
```

For instance, if you need to declare parameters of different data types, your function definition may appear as follows:

```
function_name( int lower, int upper, int temp, char x, float y )
{
    .
    .
    .
}
```

If you are using the function prototype format in a function definition, you must supply both an identifier and a data-type specification for each parameter. If you do not, the action generates an error message.

In a function definition, you have the following two options when specifying an empty parameter list:

- You can specify empty parentheses.
- You can use the keyword **void** to specify an empty parameter list.

The following example shows the use of the **void** keyword:

```
char function_name( void )
{ return 'a'; }
```

5.1.3 Function Return Data Types

By default, all VAX C functions return objects of data type **int**. In Example 5-1, function `lower` returns an integer to the main function using the **return** statement.

If you define a function that returns anything other than an integer, you need to specify the function return data type in the function definition. The following example shows the definition of a function returning a character:

```
char letter( int param1, char param2, int *param3)
{
    .
    .
    .
    return param2;
}
```

If a function does not return a value, or if you do not call the function within an expression that requires a value, you can define the function as type **void**. Using the **void** keyword in a function declaration generates an error under the following conditions:

- If the function returns a value
- If you call the **void** function in an expression that requires a return value
- If you use the cast operator to cast anything other than a function to the **void** type

The following example shows how to use the **void** keyword to specify a function without a return value and to specify a null parameter list:

```
void message( void )
{
    printf("Stop making sense!");
    return;
}
```

5.1.4 Variable-Length Parameter Lists

If you decide to define a function with a variable-length parameter list, you can use ellipses (`...`) in a function prototype declaration to designate the variable-length portion of the parameter list, as follows:

```
function_name(int lower, int upper, char x, float y, ... )
{
    .
    .
    .
}
```

Within the function body, use the `stdarg` functions and macros to access the argument list passed to the function. The `stdarg` functions and macros provide a portable means of accessing variable-length argument lists. For more information about variable-length argument lists, see the `stdarg` information in the *VAX C Run-Time Library Reference Manual*.

When using ellipses for variable-length argument lists, you must have at least one argument preceding the ellipses. The following definition is legal:

```
function_name( double lower, ... )
{
    ...
}
```

The following definition is not legal:

```
function_name( ... )
{
    .
    .
    .
}
```

If you are not using function prototypes, you can use the **`stdarg`** header and declaration within the parameter list and before the function body, as opposed to using the ellipsis notation. The following example shows such a construct:

```

function_name( lower, upper, x, y, va_alist )
int lower, upper;
char x;
float y;
va_dcl
{
    .
    .
    .
}

```

NOTE

If you are using function prototypes, use ellipses (. . .) within parameter lists so that the compiler does not compare varargs declarations (va_alist, va_dcl) with prototype data declarations. See Section 5.3 for more information about function prototypes.

5.2 Function Declarations

As in Example 5–1, you may call a function without declaring it if the function’s return value is an integer. If the return value is anything else, the function may have to be declared. Example 5–2 shows when you need to declare a function.

Example 5–2: Declaring Functions

```

main()
{
    ❶ char lower();          /* The function declaration */
    .
    .
    .
    while ((c = getc(infile)) != EOF)
    {
        /* The function call */
        c_out = lower(c);
        putc(c_out, outfile);
    }
}

```

(continued on next page)

Example 5-2 (Cont.): Declaring Functions

```
char lower(c_up)          /* The function definition */
int c_up;
{
    .
    .
    .
}
```

Key to Example 5-2:

- ① Since the location of the function definition is after the main function in the source code, and since function `lower` has a return type of **char**, you have to declare the function before calling it.

If the function definition of `lower` was located before the main function in the source code, you would not have to declare function `lower` before calling the function.

In a function declaration, you can use the **void** keyword to specify an empty argument list, as follows:

```
main()
{
    char function_name( void );
    .
    .
    .
}
char function_name( void )
{ }
```

If the function does not return a value, you can use the **void** keyword in the declaration and definition, as follows:

```
main()
{
    void function_name( );
    .
    .
    .
}
void function_name( )
{ }
```

If you specify argument data types or **void** in the parameter list of a function declaration, as shown in the following example, VAX C treats the function declaration as a function prototype for the scope of the declaration:

```
main()
{
    char function_name( int x, char y );
    .
    .
    .
}
```

Since the declaration is within the scope of function main, VAX C uses the function declaration as a function prototype only within function main. See Section 5.3 for more information about function prototypes.

5.3 Function Prototypes

A *function prototype* is a function declaration that specifies the data types of its arguments in the identifier list. VAX C uses the prototype to ensure that all function definitions, declarations, and calls within the scope of the prototype contain the correct number of arguments or parameters, and that each argument or parameter is of the correct data type.

Function prototypes provide argument checking found in the LINT utility provided with other implementations of C. See Section 5.9 for more information.

When using function prototypes, you can first define the following function:

```
char function_name( int lower, int *upper, char (*func)(), double y )
{ }
```

You can also use the following code:

```
char function_name( lower, upper, func, y )
int lower;
int *upper;
char (*func)();
double y;
{ }
```

This function's identifier list includes an integer, a pointer to an integer, a pointer to a function returning a character, and a double floating-point value. The type specifications are identical to the ones used in a parameter list located before the function body. For more information about interpreting complex declarations, see Section 8.12.

In each compilation unit in your program, determine where to place the corresponding function prototype. The position of the prototype determines the prototype's scope; the scope of the function prototype is the same as the scope of any function declaration. VAX C checks all function definitions, declarations, and calls from the position of the prototype to the end of its scope. If you misplace the prototype so that a function definition, declaration, or call occurs outside the scope of the prototype, the results are undefined.

Corresponding function prototype declarations are identical to the header of a function definition that specifies data types in the identifier list. Since prototypes are function declarations, you end the prototype code with a semicolon (;). The following code example is a prototype that corresponds with either of the previous function definitions:

```
char function_name( int lower, int *upper, char (*func)(), double y );
```

When declaring nondefinition function prototypes, you do not need to use the same parameter identifiers as in the function definition. If you choose, you do not need to specify any identifiers in the prototype declaration. The scope of the identifiers within function prototypes exists only within the identifier list; you are free to use those identifiers outside the prototype.

For example, you can use any of the following prototype declarations for the function definition presented:

```
char function_name( int lower, int *upper, char (*func)(), double y );
char function_name( int a, int *b, char (*c)(), double d );
char function_name( int, int *b, char (*c)(), double );
char function_name( int, int *, char (*)(), double );
```

You can specify variable-length argument lists in function prototypes by using ellipses. You must have at least one argument in the list preceding the ellipses. The following example shows the specification of a variable-length argument list:

```
char function_name( int lower, . . . );
```

You cannot omit data-type specifications in a function prototype. Also, you cannot have a variable-length argument list that is not preceded by at least one argument. The following prototypes are not legal and their use generates error messages:

```
char function_name( lower, *upper, char (*func)(), float y );
char function_name( , , char (*func)(), float y );
char function_name( . . . );
```

5.3.1 Using Function Prototypes

Using function prototype ensures that all corresponding function definitions, declarations, and calls within the scope of the prototype conform to the number and type of parameters specified in the prototype. A function prototype is considered in scope only if a function prototype declaration is specified within a block enclosing the function call or at the outermost level of the source file. If a prototype is in scope, the automatic widening of **float** arguments to **double** is not performed. However, the automatic widening of **char** and **short int** arguments to **int** is performed. If the number of arguments in a function definition, declaration, or call does not match the prototype, the statement generates the appropriate message.

If the data type of an argument in a function call does not match the prototype, VAX C attempts to perform conversions. If the mismatched argument is assignment compatible with the prototype parameter, VAX C converts the argument to the data type specified in the prototype, according to the parameter and argument conversion rules (see Section 5.4).

If the mismatched argument is not assignment compatible with the prototype parameter, the action generates the appropriate error message and the results are undefined.

The syntax of the function prototype is designed so that you can extract the first line of each of your function definitions, add a semicolon (;) to the end of each line, place the prototypes in a .H definitions file, and include that file at the top of each compilation unit in your program. In this way, you declare the function prototypes to be external, so that the scope of the prototype extends throughout the entire compilation unit. To use prototype checking for VAX C Run-Time Library (RTL) function calls, include the module or modules appropriate for the VAX C RTL functions used in your program. You place the include preprocessor directives at the top of any applicable compilation units.

For basic descriptions of the VAX C RTL prototype include modules, see Appendix A. For more information about the **#include** preprocessor directives, see Section 10.4. For more information about compilation units and scope, see Section 9.1.

5.4 Using Parameters and Arguments

VAX C functions can exchange information by means of parameters and arguments. (In this guide, the term *parameter* denotes the variable within parentheses named in a function definition; the term *argument* denotes an expression that is part of a function call.) In Example 5-1, function `lower` has the single parameter `c_up`. When this function is called from the main function, argument `c` is evaluated and passed to function `lower`.

The following rules apply to parameters and arguments of VAX C functions:

- The number of arguments in a function call must be the same as the number of parameters in the function definition. This number may be zero.
- In VAX C, the maximum number of arguments (and corresponding parameters) is 253 for a single function. The maximum length of an argument list is 255 longwords.
- Arguments are separated by commas. However, the comma is not an operator in this context, and the arguments may be evaluated by the compiler in any order. Do not expect function calls or other complicated expressions in the argument list to be evaluated in any particular order.
- In VAX C, arguments are passed by value; that is, when a function is called, the parameter receives a copy of the argument's value, not its address. This rule applies to all scalar variables, structures, and unions passed as arguments. Function and array names used as arguments undergo conversions that are described later in this list.
- A function cannot modify the values of its arguments. However, since arguments can be addresses or pointers, a function can use addresses to modify the values of variables defined in the calling function.

NOTE

When passing arguments between programs written in VAX C and programs written in other VMS programming languages, remember the restrictions of the VAX Procedure and Condition Handling Standard (sometimes called the VAX Calling Standard). For more information about the VAX Calling Standard and passing arguments in VAX C, see Section 13.1.

- The types of evaluated arguments must match the types of their corresponding parameters. When a function is called, unless a function prototype is in scope, VAX C does not compare the types of the arguments with those of the corresponding parameters; so it does not generally convert the arguments to the types of the parameters. Instead,

all the expressions in the argument list are converted according to the following conventions:

- Any arguments of type **float** are converted to **double**.
- Any arguments of type **char** or **short** are converted to **int**.
- Any arguments of type **unsigned char** or **unsigned short** are converted to **unsigned int**.
- Any function name appearing as an argument is converted to the address of the named function. You must declare the corresponding parameter as a pointer to a function, which evaluates to a value of the same data type as the function.
- Any array name appearing as an argument is converted to the address of the first element of the array. You must declare the corresponding parameter either as an array of the given type or as a pointer to the given type. Since character-string constants are declared implicitly as arrays of characters, this rule also applies to the use of string constants as arguments.

No other default conversions are performed on arguments. If you know that a particular argument must be converted to match the type of the corresponding parameter, use the cast operator. For more information about the cast operator, see Section 7.4.5.

- If you declare variables in the parameter declaration section that do not exist in the parameter list, these variables are treated as if they were declared in the function body. However, this is not good programming practice and, if used, your programs may not be portable.
- If you do not declare parameters, they are implicitly declared to be of data type **int**.

The following sections discuss the following topics:

- Function and array identifiers as arguments
- Passing arguments to the main function

5.4.1 Function and Array Identifiers as Arguments

You can use a function identifier without parentheses and arguments. In this case, the function identifier evaluates to the address of the function. This method of referencing is useful when passing a function identifier in an argument list. You can pass the address of one function to another as one of the arguments.

If you wish to pass the address of a function in an argument list, the function must either be declared or defined, even if the return value of the function is an integer. Example 5-3 shows when you must declare user-defined functions and how to pass functions as arguments.

Example 5-3: Declaring Functions Passed as Arguments

```

❶ x() { return 25; }

int z[10];

main()
{
❷ int y(); /* Function declaration */
  .
  .
❸ funct(x, y, z); /* Passed as addresses */
  .
  .
}

y() { return 30; } /* Function definition */

funct(f1, f2, a) /* Function definition */
/* Declare arguments as */
/* pointers to functions */
/* returning an integer */

int *a;
❹ int (*f1)(), (*f2)();
  {
    (*f1)(); /* A call to a function */
    .
    .
  }

```

Key to Example 5-3:

- ❶ You can pass function x in an argument list, since its definition is located before the main function.
- ❷ You must declare function y before you pass the function in an argument list, since its function definition is located after the main function.
- ❸ When you pass functions as arguments, do not include the parentheses. Similarly, when you specify arrays, do not include subscripts.

- ④ When declaring parameters that represent functions, declare them as pointers to functions. When declaring arrays, declare the parameter as a pointer to the type of the array. For convenience, declarations of parameters, which are functions or arrays, can be declared as ordinary function or array declarators; the compiler automatically converts them to pointers.

5.4.2 Passing Arguments to the main Function

The main function in a VAX C program can accept arguments from the command line from which it was invoked. The syntax for a main function is as follows:

```
int main(argc, argv, envp)

int argc;
char *argv[ ],*envp[ ];
```

argc

Is the number of arguments present in the command line that invoked the program.

argv

Is a character-string array of the arguments.

envp

Is the environment array. This array contains process information, such as the user name and controlling terminal. It has no bearing on passing command-line arguments. Its primary use in VAX C programs is during **exec** and **getenv** function calls. (See the *VAX C Run-Time Library Reference Manual* for more information).

In the main function definition, the parameters are optional. However, you can access only the parameters that you define. You can define function main in any of the following ways:

```
main()
main(argc)
main(argc, argv)
main(argc, argv, envp)
```

To pass arguments to the main function, you must install the program as a DCL foreign command. When a program is installed and run as a foreign command, the parameter `argc` is greater than or equal to 1, and `argv[0]` contains the name of the image file.

The procedure for installing a foreign command involves using a DCL assignment statement to assign the name of the image file to a symbol that is later used to invoke the image. For example:

```
$ ECHO == "$ DSK$:COMMARG.EXE" RETURN
```

The symbol ECHO is installed as a foreign command that invokes the image in COMMARG.EXE. The definition of ECHO must begin with a dollar sign (\$) and include a device name, as shown.

For more information about the procedure for installing a foreign command, see the *VMS DCL Dictionary*.

Example 5-4 shows a program called COMMARG.C, which displays the command-line arguments that were used to invoke it.

Example 5-4: Echo Program Using Command-Line Arguments

```
/* This program echoes the command-line arguments.          */
#include stdio
main(argc, argv)
int  argc;
char *argv[];
{
    int i;
                                /* argv[0] is program name */
    printf("program: %s\n",argv[0]);
    for (i = 1; i < argc; i++)
        printf("argument %d: %s\n", i, argv[i]);
}
```

You can compile and link the program using the following DCL command lines:

```
$ CC COMMARG RETURN
$ DEFINE LNK$LIBRARY SYS$LIBRARY:VAXCRT.LIB RETURN
$ LINK COMMARG RETURN
```

A sample output for Example 5–4 is as follows:

```
$ ECHO Long "Day's" "Journey into Night" [RETURN]
program: db7:[oneill.plays]commarg.exe;1
argument 1: long
argument 2: Day's
argument 3: Journey into Night
```

DCL converts most arguments on the command line to uppercase letters. However, VAX C internally parses and modifies the altered command line to make VAX C argument access compatible with C programs developed on other systems.

All alphabetic arguments in the command line are delimited by spaces or tabs. Arguments with embedded spaces or tabs must be enclosed in quotation marks (" "). Uppercase characters in arguments are converted to lowercase, but arguments within quotation marks are left unchanged.

5.5 Identifiers

Identifiers can consist of letters, digits, dollar signs (\$), and the underscore character (_). Do not create identifiers with a length of more than 255 characters. If you do, the compiler ignores all characters after the two hundred and fifty-fifth character. If the identifier will be seen by the linker, as in a declaration with [extern] or #*module*, do not use more than 31 characters.

The first character must not be a digit and, to avoid conflict with names used by VAX C, should not be an underscore character. VAX C uses a preceding underscore to identify implementation-specific macros and keywords, and uses two preceding underscores to identify implementation-specific constants.

Upper- and lowercase letters specify different variable identifiers; that is, the compiler interprets abc and ABC as different variable names.

Use the dollar sign only within identifiers for VMS global symbols. Identifiers that contain dollar signs may not be portable.

DIGITAL recommends the following conventions if practical:

- Avoid using underscores as the first character of your identifiers.
- Type identifiers in uppercase if they are constants that are given values by the #**define** directive.

- Type all instances of a global name in the same case. All names that become part of the VMS Linker's global symbol table are represented there in uppercase. Consider these examples:

```
int globalvalue ss$_accvio = 0;  
globalvalue SS$_ACCVIO;
```

The compiler will consider these to denote different global names; however, uppercase forms for both are passed to the linker, potentially causing errors when the program is linked or executed. For more information about **globalvalue**, see Section 9.6.2.

- Type all other identifiers and keywords in lowercase.

5.6 Language Keywords

The VAX C keywords are predefined identifiers. They cannot be redeclared. They identify data types, storage classes, and certain statements in VAX C. Note that many conventional words in VAX C programs are not keywords and can be redeclared. The notable examples are the names of functions, including `main` and the functions found in libraries that accompany the VAX C compiler.

Keywords must be expressed in lowercase letters.

Table 5-1 lists the VAX C keywords and their meaning.

Table 5–1: VAX C Keywords

Keyword	Meaning
Type specifiers:	
int	Integer (On VAX systems, 32 bits)
long	32-bit integer
unsigned	Unsigned integer
short	16-bit integer
char	8-bit integer
float	Single-precision, floating-point number
double	Double-precision, floating-point number
struct	Structure (aggregate of other types)
union	Union (aggregate of other types)
typedef	Tagged set of type specifiers
enum	Enumerated scalar type
void	Function return type
variant_struct	Variant structure
variant_union	Variant union
Data-type modifiers:	
const	Definition of constant data
volatile	Definition of volatile data
Storage-class specifiers:	
auto	Allocated at every block activation
static	Allocated at compile time
register	Allocated at every block activation
extern	Allocated by an external data definition (at compile time)
globaldef	Definition of a global variable
globalref	Reference to a global variable
globalvalue	Definition or declaration of a global value
readonly	Allocated in read-only program section
noshare	Assigned NOSHR program section attribute
_align	Aligns data on specific storage boundaries

(continued on next page)

Table 5–1 (Cont.): VAX C Keywords

Keyword	Meaning
Statements:	
goto	Transfers control unconditionally
return	Terminates a function and optionally returns a value to the caller
continue	Causes next iteration of a containing loop
break	Terminates its corresponding switch or loop
if	Executes the following statement conditionally
else	Provides an alternative for the if statement
for	Iterates the next statement (zero or more times) under control of three expressions
do	Iterates the next statement (one or more times) until a given condition is false
while	Iterates the next statement (zero or more times) while a given expression is true
switch	Executes one or more of the specified cases (multiway branch)
case	Begins one case for switch
default	Provides default case for switch
Operator:	
sizeof	Computes the size of an operand, in bytes

Although they are not true keywords, the VAX C compiler defines substitutions for the following identifiers; you should avoid redefining them:

```
vms          VMS
vax          VAX
vaxc        VAXC
vax11c      VAX11C
vms_version VMS_VERSION

CC$gfloat
CC$parallel
```

For more information about these identifiers, see Section 11.1.7.

5.7 Blocks

A *block* is a compound statement surrounded by braces ({}). You can use a block when the grammar of VAX C requires a single statement. The common cases are the bodies of functions and **if**, **for**, **do**, **switch**, and **while** statements. Note that this definition of a block may conflict with its definition in other languages. In VAX C, the terms block and compound statement are equivalent.

A block may also contain declarations. If it does, any declarations of **auto**, **register**, or **static** variables declare names that are local to the block. Example 5-5 presents nested blocks and the differences in the scope of declared variables.

Example 5-5: Scope of Variable Declarations in Nested Blocks

```
/* This program shows how variables with the same      *
 * identifier can be of different data types if located *
 * in different blocks.                                */
main()
{
    /* Outer block of "main"      */
    ① int i;
      i = 1;
      .
      .
      .
      if (i == 1)
      {
          /* An inner block      */
          ② float i;
            .
            .
            .
            i = 3e10;
        }
    }
}
```

Key to Example 5-5:

- ① In all blocks of the program, except the block in the **if** statement, variable **i** is an integer. The default storage class for this variable is **auto**.

- ② Within the block in the **if** statement, variable **i** is a single-precision, floating-point value. Since it is also of the storage class **auto**, a new floating-point version of variable **i** is allocated each time the inner block is activated.

If initialization is specified for any **auto** or **register** variables in a block, it is performed each time control reaches the block normally; that is, such initializations are not performed if a **goto** statement transfers control into the middle of the block or if the block is the body of a **switch** statement. For more information about data types, see Chapter 8. For more information about scope and storage classes, see Chapter 9.

5.8 Comments

Comments, delimited by the character pairs `(/*` and `*/)`, can be placed anywhere that white space can appear. The text of a comment can contain any characters except the close-comment delimiter `*/`. Comments cannot be nested.

5.9 LINT-Like Functionality

Some implementations of C provide a utility called LINT. LINT provides a way to check source code for improper definitions and declarations, for parameter and argument mismatching, and for inefficient coding practices. VAX C provides the following features shown in Table 5-2 that, in combination, offer much of the functionality of LINT.

Table 5–2: VAX C Features Similar to the LINT Utility

Feature	Description
<code>/STANDARD=PORTABLE</code>	When you compile your source code, add this qualifier to <code>CC</code> . The compiler flags constructs that may not be supported by other implementations of the C language.
Function prototypes	The use of function prototypes allows VAX C to check the number and the data types of all arguments passed to functions. See Section 5.3 for complete information.
SCA support	The VAX Source Code Analyzer (SCA) is a source code cross-reference and static analysis tool that you can use with VAX C source code. SCA's query and reporting facilities allow you to query a library for the presence of specific symbol, file, or module information, and to discern such things as declarations of program symbols, references to the symbols, and references to the source files.

Chapter 6

Statements

This chapter describes the statements in the VAX C programming language. Statements are executed in the sequence in which they appear in a program, except as indicated. The VAX C statements are grouped as follows:

- Control flow statements (Section 6.1)
- Expressions and blocks as statements (Section 6.2)
- Conditional statements (Section 6.3)
- Looping statements (Section 6.4)
- Interrupting statements (Section 6.5)

6.1 Control Flow Statements

You can use some VAX C statements either to maintain or modify the control of the program. The following sections describe the control flow statements.

6.1.1 The null Statement

Use null statements to provide null operations in situations where the grammar of the language requires a statement, but the program requires no work to be done.

The syntax of the null statement is as follows:

;

You may need to use the null statement with the **if**, **while**, **do**, and **for** statements in cases where the grammar requires a statement body but the program requires no functional operation. The most common use of this statement is in loop operations, where all the loop activity is performed by the test portion of the loop. For example, the following statement finds the first element of an array known to have a value of zero:

```
for(i=0; array[i] != 0; i++)  
    ;
```

See Section 6.2 and Section 6.4 for more information about the statements mentioned here.

6.1.2 The goto Statement

The **goto** statement transfers control unconditionally to a label statement, where the label identifier must be located in the scope of the function containing the **goto** statement.

The syntax of the **goto** statement is as follows:

```
goto identifier;
```

Take care when branching into a block or function body using the **goto** statement. The compiler allocates storage for automatic variables declared within a block when the block is activated. When a **goto** statement branches into a block, automatic variables declared in the block cannot exist in storage. Attempts to access such variables can cause a run-time error.

6.1.3 The label Statement

Labels are identifiers used to flag a location in a program, and to be the target of a **goto** statement.

The syntax of a label is as follows:

```
identifier:
```

Any statement can be preceded by a label. The scope of the label is the current function body. Variables can have the same name as the label in the function because the label name is independent of the scope rules applied to variables. Labels are used only as the targets of **goto** statements.

6.2 Expressions and Blocks as Statements

The statements in the following sections are expressions or groups of other statements that you can use when the grammar calls for a single statement.

6.2.1 The expression Statement

You can use any valid expression as a statement by terminating it with a semicolon. The following example is an expression used as a statement:

```
i++;
```

This statement increments the value of the variable `i`. Note that `i++` is a valid VAX C expression that can appear in more complex VAX C statements. For more information about the valid VAX C expressions, see Section 7.2.

6.2.2 The compound Statement

A compound statement in VAX C is often called a block. It allows more than one statement to appear where a single statement is required by the language. The following code is an example of a compound statement:

```
{
    int x = 5;
    z = 1;
    if (y < x)
        funct(y, z);
    else
        funct(x, z);
}
```

The compound statement contains optional declarations followed by a list of statements, all enclosed in braces. If you include declarations, the variables they declare are local to the block, and, for the rest of the block, they supersede any previous declaration of variables of the same name. Inside blocks, you can initialize variables whose declarations include the **auto**, **register**, **static**, or **globaldef** storage-class specifiers.

A block is entered “normally” when control flows into it, or when a **goto** statement transfers control to a label on the block itself. The compiler-generated code allocates storage for **auto** or **register** variables each time the block is entered normally; the storage allocations do not occur if a **goto** statement refers to a label inside the block or if the block is the body of a **switch** statement. For more information about storage classes, see Chapter 9.

All function definitions are compound statements. The compound statement following the parameter declarations in a function definition is called the function body.

6.3 Conditional Statements

The statements in the following sections execute only if a tested condition is true.

6.3.1 The if Statement

An **if** statement executes a statement depending on the evaluation of an expression, and may or may not be written with an **else** clause.

The syntax of the **if** statement is as follows:

```
if ( expression )
    statement
else
    statement
```

An example of the **if** statement is as follows:

```
if ( i < 1 )
    funct (i);
else
{
    i = x++;
    funct (i);
}
```

If the evaluated expression within parentheses is true (in the example, if variable *i* is less than 1), then the statement following the evaluated expression executes; the statement following the keyword **else** does not execute. If the evaluated expression is false, then the statement following the keyword **else** executes.

All logical operators define a true result to be nonzero. Therefore, the expression in any conditional statement can be a logical expression with predictable results (true or false; nonzero or zero).

When **if** statements are nested within **else** clauses, an **else** clause matches the most recent **if** statement that does not have an **else** clause.

6.3.2 The switch Statement

The **switch** statement executes one or more of a series of cases, based on the value of the expression.

The syntax of the **switch** statement is as follows:

```
switch ( expression )  
    statement
```

The usual arithmetic conversions are performed on the expression, but the result must be type **int**. For more information about data type conversion, see Section 7.9. The statement is typically a compound statement, within which one or more **case** labels prefix statements that execute if the expression matches the **case**.

The syntax for a **case** label and expression follows:

```
case constant-expression :  
    statement[,statement, . . . ]
```

The constant expression must also be of type **int**. No two **case** labels can specify the same value. The value of a constant expression can be any integral value.

Only one statement in the compound statement can have the following label:

```
default :
```

The **case** and **default** labels can occur in any order. Note that each case flows into the next unless explicit action is taken, such as a **break** statement. When the **switch** statement is executed, the following sequence takes place:

1. The **switch** expression is evaluated and compared with the constant expressions in the **case** labels.
2. If the expression's value matches a **case** label, the statements following that label are executed. If the list of statements ends with the **break** statement, the **break** terminates the **switch** statement; otherwise, the next case encountered is executed. (See Example 6–1.) The **switch** statement can also be terminated by a **return** or **goto** statement; if the **switch** is inside a loop, it can be terminated by a **continue** statement. For more information about interrupting statements, see Section 6.5.
3. If the expression's value does not match any **case** label but there is a **default** case, the **default** case is executed. It need not be the last case listed. If a **break** statement does not end the **default** case and it is not the last case, the next case encountered is executed.

4. If the expression's value does not match any **case** label and there is no **default**, the body of the **switch** statement is not executed.

In general, the **break** statement must be used to ensure that a **switch** statement executes as expected. Example 6–1 uses the **switch** statement to count blanks, tabs, and newlines entered from the terminal.

Example 6–1: Using switch to Count Blanks, Tabs, and Newlines

```
/* This program counts blanks, tabs, and newlines in text *
 * entered from the keyboard. */

#include stdio
main()
{
    int number_tabs = 0, number_lines = 0, number_blanks = 0;
    int ch;
    while ((ch = getchar()) != EOF)
        switch (ch)
        {
            ① case '\t': ++number_tabs;
            ②         break;
            case '\n': ++number_lines;
                    break;
            case ' ': ++number_blanks;
                    break;
        }
    printf("Blanks\tTabs\tNewlines\n");
    printf("%6d\t%6d\t%6d\n", number_blanks,
        number_tabs, number_lines);
}
```

Key to Example 6–1:

- ① A series of **case** labels is used to increment the counters.
- ② The **break** statement causes control to go back to the **while** loop every time a counter increments. The program automatically passes control to the **while** loop if none of the counters is incremented.

The program in Example 6–1 responds to the following input:

```
$ RUN EXAMPLE.EXE [RETURN]
Every good boy.[RETURN]
The quick brown fox.[RETURN]
Line with 2 [TAB][TAB]tabs.[RETURN]
^Z
```

Example 6-1 produces the following output:

```
Blanks      Tabs  Newlines
   7         2      3
```

If you omit the **break** statements, the program prints the following:

```
Blanks      Tabs  Newlines
  12         2      5
```

Without the **break** statements, each case drops through to the next case. The number shown for tabs happens to be right, because the tabs case is first in the **switch** statement and is executed only if `ch == '\t'`. Notice that the number shown for newlines is the correct number plus the number of tabs, and the number shown for blanks is the total of all three cases.

6.3.2.1 Declarations Within a switch Statement

If variable declarations appear in the compound statement within a **switch** statement, any initializations of **auto** or **register** variables are ineffective. However, if you initialize variables within the statements following a **case** label, the initialization is effective. Consider the following example:

```
switch (ch)
{
    int x = 1;          /* Improper initialization */
    printf("%d", x);   /* This first printf won't be executed */
    case 'a' :
        { int x = 5;   /* Proper initialization */
          printf("%d", x);
          break; }
    case 'b' :
        .
        .
        .
}
```

In the previous example, if the variable `ch` equals `'a'`, then the program prints the value 5. If the variable equals any other letter, the program prints nothing because the initialization outside of the case label is ineffective.

6.4 Looping Statements

The statements in the following sections execute repeatedly (loop) until an expression evaluates to false. Some loops execute a block of statements, known as the loop body, a specified number of times (in VAX C, the **for** statement); some loops evaluate an expression and then execute the body of the loop (in VAX C, the **while** statement); some loops execute the loop body and then evaluate the expression, which guarantee at least one execution of

the body (in VAX C, the **do** statement). The following sections discuss the **for**, **while**, and **do** statements.

6.4.1 The for Statement

The **for** statement evaluates three expressions and executes a statement (the loop body) until the second expression evaluates to false. The **for** statement is useful for executing a loop body a specified number of times.

The syntax for the **for** statement is as follows:

```
for ( expression-1 ; expression-2 ; expression-3 )
    statement;
```

The **for** statement executes the loop body zero or more times. It uses three control expressions as shown. Semicolons (;) are used to separate the expressions; notice that a semicolon does not follow the last expression. A **for** statement executes the following steps:

1. Expression-1 is evaluated only once before the first iteration of the loop. It usually specifies the initial values for variables.
2. Expression-2 is a relational or logical expression that determines whether or not to terminate the loop. Expression-2 is evaluated before each iteration. If the expression evaluates to false, execution of the **for** loop body terminates. If the expression evaluates to nonzero, the body of the loop is executed.
3. Expression-3 is evaluated after each iteration. It usually specifies increments for the variables initialized by expression-1.
4. Iterations of the **for** statement continue until expression-2 produces a false (zero) value, or until some statement—such as **break** or **goto**—interrupts.

The **for** statement is equivalent to the following code:

```
expression-1;
while ( expression-2 )
{
    statement
    expression-3;
}
```

The VAX C compiler optimizes certain **for** statements for simple loops such as the following example:

```
for(i=0; i<15; i++)
    printf("%d\n", i);
```

When the incrementation is as simple as in the previous example, the compiler generates less macro code so efficiency increases. When possible, use **for** statements as opposed to **while** statements when the increment is small.

Any of the three expressions in a loop can be omitted. If expression-2 is omitted, the test condition is true; that is, the **while** in the expansion becomes **while(x)**, where x is not equal to zero. If either expression-1 or expression-3 is omitted from the **for** statement, that expression is effectively dropped from the expansion.

The following syntax shows an infinite loop:

```
for (;;) statement
```

Terminate infinite loops with a **break**, **return**, or **goto** statement.

6.4.2 The while Statement

The **while** statement evaluates an expression and executes a statement (the loop body) zero or more times, until the expression evaluates to false.

The syntax of a **while** statement is as follows:

```
while ( expression )  
    statement
```

An example of the **while** loop is as follows:

```
while (x < 10)  
{  
    array[x] = x;  
    x++;  
}
```

This statement tests the value of the variable x; if variable x is less than 10, it assigns x to the xth element of the array and then increments the variable x. If the expression in parentheses evaluates to false, the loop body never executes.

6.4.3 The do Statement

The **do** statement executes a statement (the loop body) one or more times, until the expression in the **while** clause evaluates to false.

The syntax for the **do** statement is as follows:

```
do
    statement
while ( expression ) ;
```

The statement is executed at least once, and the expression is evaluated after each subsequent execution of the loop body. If the expression is true, the statement is executed again.

6.5 Interrupting Statements

You can use the statements in the following sections to interrupt the execution of another statement. These statements are primarily used to interrupt **switch** statements and loops.

6.5.1 The break Statement

The **break** statement terminates the immediately enclosing **while**, **do**, **for**, or **switch** statement. Control passes to the statement following the loop body.

The syntax for the **break** statement is as follows:

```
break;
```

6.5.2 The continue Statement

The **continue** statement passes control to the end of the immediately enclosing **while**, **do**, or **for** statement.

The syntax for the **continue** statement is as follows:

```
continue;
```

The **continue** statement is equivalent to the **goto** label statement, shown here, for each of the looping statements in the syntax examples that follow:

```

while( . . . )      do      for( . . . ; . . . ; . . . )
{
    .
    .
    goto label;
    .
    .
label:
    ;
}
                    {
    .
    .
    goto label;
    .
    .
label:
    ;
}
                    {
    .
    .
    goto label;
    .
    .
label:
    ;
}
                    while( . . . );

```

In the preceding syntax examples, a **continue** statement passes control to label. The **continue** statement is intended only for loops, not for **switch** statements. A **continue** inside a **switch** statement that is inside a loop causes continued execution of the enclosing loop after exiting from the body of the **switch** statement.

6.5.3 The return Statement

The **return** statement causes a return from a function, with or without a return value.

The syntax of the **return** statement is as follows:

```
return [expression];
```

The compiler evaluates the expression (if you specify one) and returns the value to the calling function. If necessary, the compiler converts the value to the declared type of the containing function's return value. If there is no specified return value, the value is undefined.

You can declare a function without a **return** statement to be of type **void**. For more information about the **void** data type and function return values, see Section 5.2.

Expressions and Operators

An *expression* is any series of symbols that VAX C uses to produce a value. The simplest expressions are constants and variable names, which yield a value directly. Other expressions combine operators and subexpressions to produce values.

In some instances, the compiler makes conversions so that the data types of the operands are compatible. This chapter refers to these rules as the *arithmetic conversion rules*. See Section 7.9.1 for more information about these rules.

This chapter discusses the following topics:

- lvalues and rvalues (Section 7.1)
- Primary expressions and operators (Section 7.2)
- An overview of the VAX C operators (Section 7.3)
- Unary expressions and operators (Section 7.4)
- Binary expressions and operators (Section 7.5)
- The conditional expression and operator (Section 7.6)
- Assignment expressions and operators (Section 7.7)
- The comma expression and operator (Section 7.8)
- Data-type conversions (Section 7.9)

7.1 lvalues and rvalues

A variable identifier is one of the primary VAX C expressions. (See Section 7.2 for more information about primary expressions.) This type of expression yields a single value. However, when using the variable identifier with other operators, the expression evaluates to the variable's location in memory. The address of the variable is the variable's lvalue. The object stored at that address is the variable's rvalue. For example, VAX C uses both the lvalue and the rvalue of variables in the evaluation of an expression as follows:

```
x = y;
```

The contents of variable *y* are taken and assigned to variable *x*. The expression on the right side evaluates to the variable's rvalue while the expression on the left side evaluates to the variable's lvalue when performing an assignment.

The following syntax defines those VAX C expressions that either have or produce lvalues:

```
lvalue ::=
    identifier
    primary [ expression ]
    lvalue . identifier
    primary -> identifier
    * expression
    ( lvalue )
```

These expressions represent, respectively:

- Identifiers of scalar variables, structures, and unions
- References to scalar array elements
- References to structure and union members, except for references to fields that are not lvalues
- Indirect references to structure and union members, except for references to fields that are not lvalues
- References to pointers (also called dereferenced pointers; an asterisk (*) followed by an address-valued expression)
- Any of the previous expressions, enclosed in parentheses

All lvalue expressions represent a single location in a computer's memory.

7.2 Primary Expressions and Operators

Simple expressions are called *primary expressions*; they denote values. Primary expressions include previously declared identifiers, constants (including strings), array references, function calls, and structure or union references.

The syntax descriptions of the primary expressions are as follows:

```
primary ::=
    identifier
    constant
    string
    ( expression )
    primary ( expression-list )
    primary [ expression ]
    lvalue . identifier
    primary -> identifier
```

The simplest forms are identifiers such as variable names and string or arithmetic constants. Other forms are expressions (delimited by parentheses), function calls, array references, lvalues and rvalues, and structure and union references.

The following sections describe the primary expressions and operators.

7.2.1 Parenthetical Expressions

An expression within parentheses has the same type and value as the same expression without parentheses. As in declarations, any expression can be delimited by parentheses to change the grouping, or associative precedence, of the operators in a larger expression.

7.2.2 Function Calls

A *function call* is a primary expression followed by parentheses. The parentheses may contain a list of arguments (separated by commas) or may be empty. An undeclared function is assumed to be a function returning **int**. If you declare an identifier as a “function returning . . .”, but use the identifier in a context other than a function call, it converts to “the address of function returning . . .”. When you pass an argument that is an array or function, specify the identifier in the argument list. The compiler passes the address of the array or function to the called routine. This means that

the corresponding parameters in the called function must be declared as pointers. For example:

```
int fl();
.
.
.
fl();
```

Consider the following declaration:

```
double atof();
```

The previous example declares a function returning **double**. You can then use the identifier **atof** in a function call, as follows:

```
result = atof(c);
```

You can use the identifier **atof** in other contexts without the parentheses. For example:

```
dispatch(atof);
```

The identifier **atof** converts to the address of that function, and the address is passed to the function `dispatch`.

Functions can also be called using a pointer to a function. Consider the following pointer declaration and assignment:

```
double (*pfd)();
.
.
.
pfd = atof;
```

To call the function, you can specify the following form:

```
result = (*pfd)(c);
```

VAX C also accepts a pointer to a function, as shown in the following form:

```
result = pfd(c);
```

While the first call to the function is valid, the second call to the function is simpler and requires fewer keystrokes.

7.2.3 Array References

Use bracket operators (`[]`) to refer to elements of arrays. In an array defined as having three dimensions, you can refer to a specific element within the array, as in the following example:

```
int sample_array[10][5][2];          /* Array declaration      */
int i = 10;
sample_array[9][4][1] = i;          /* Assign value to element */
```

This example assigns a value of 10 to element `sample_array[9][4][1]`.

In addition, if an array reference is not fully qualified, it refers to the address of the first element in the dimension that is not specified. Consider the following statement in which the third dimension of the array is not specified:

```
sample_array[9][4] = 10;
```

This statement assigns a value of 10 to the element `sample_array[9][4][0]`. Consider the following statement in which none of the array dimensions are specified:

```
sample_array = 10;
```

This statement assigns a value of 10 to the element `sample_array[0][0][0]`. A reference to an array name with no bracket operator is often used to pass the array's address to a function, as in the following statement:

```
funcnt (array);
```

You can also use bracket operators to perform general pointer arithmetic as follows:

```
addr[intexp]
```

Here, `addr` is the address of some previously declared object (pointer-valued) and the variable, `intexp`, is an integer-valued expression. The result of the expression is scaled, or multiplied, by the size, in bytes, of the addressed object. If `intexp` is a positive integer, the result is the address of a subsequent object of this size; if `intexp` is zero, the result is the address of the same object; if `intexp` is negative, the result is the address of a previous object. The expressions `*(addr + intexp)` and `addr[intexp]` are equivalent because both expressions reference the same memory location; `*(addr + intexp)` points to the same element as `addr[intexp]`.

7.2.4 Structure and Union References

A member of a structure or union can be referenced with either of two operators: the period (.) or the right arrow (->).

A primary expression followed by a period followed by an identifier refers to a member of a structure or union and is itself a primary expression. The identifier must name a member of that structure or union. The result is a reference (if the member is a scalar) to the named member of the structure or union. The name of the desired member must be preceded by a period-separated list of the names of all higher-level members. For more information about structures and unions, see Section 8.9.

The form for a pointer to a structure and union uses the right-arrow operator. A primary expression followed by an arrow (specified with a hyphen (-) and a greater-than symbol (>)) followed by an identifier refers to a member of a structure or union. The identifier following the arrow operator must name a declared member of that structure or union. The result is a reference to the named member.

The primary expression in both cases can be either a pointer or an integer. If it is a pointer, VAX C assumes that it points to a structure where the name on the right is a member. If it is an integer, VAX C assumes that it is the absolute address of the appropriate structure in machine storage units. If you specify something other than a pointer to a structure or union, VAX C signals the QUALNOTSTRUCT informational message. If you point to a different structure or union type, VAX C signals the NONSEQUITUR informational message.

7.3 Overview of the VAX C Operators

You can use the simpler variable identifiers and constants in conjunction with VAX C operators to create more complex expressions. Table 7-1 presents the set of VAX C operators.

Table 7-1: VAX C Operators

Operator	Example	Result
- [unary]	-a	Negative of a
* [unary]	*a	Reference to object at address a
& [unary]	&a	Address of a
~	~a	One's complement of a
++ [prefix]	++a	The value of a after increment
++ [postfix]	a++	The value of a before increment
-- [prefix]	-- a	The value of a after decrement
-- [postfix]	a --	The value of a before decrement
sizeof	sizeof(t1)	Size in bytes of type t1
	sizeof e	Size in bytes of expression e
(type-name)	(t1)e	Expression e, converted to type t1
+	a + b	a plus b
- [binary]	a - b	a minus b
* [binary]	a * b	a times b
/	a / b	a divided by b
%	a % b	Remainder of a/b (a modulo b)
>>	a >> b	a, right-shifted b bits
<<	a << b	a, left-shifted b bits
<	a < b	1 if a < b; 0 otherwise
>	a > b	1 if a > b; 0 otherwise
<=	a <= b	1 if a <= b; 0 otherwise
>=	a >= b	1 if a >= b; 0 otherwise
==	a == b	1 if a equal to b; 0 otherwise
!=	a != b	1 if a not equal to b; 0 otherwise
& [binary]	a & b	Bitwise AND of a and b
	a b	Bitwise OR of a and b
^	a ^ b	Bitwise XOR (exclusive OR) of a and b
&&	a && b	Logical AND of a and b (yields 0 or 1)
	a b	Logical OR of a and b (yields 0 or 1)
!	!a	Logical NOT of a (yields 0 or 1)
?:	a ? e1 : e2	Expression e1 if a is nonzero; Expression e2 if a is zero

(continued on next page)

Table 7-1 (Cont.): VAX C Operators

Operator	Example	Result
=	a = b	b (assigned to a)
+=	a += b	a plus b (assigned to a)
-=	a -= b	a minus b (assigned to a)
*=	a *= b	a times b (assigned to a)
/=	a /= b	a divided by b (assigned to a)
%=	a %= b	Remainder of a/b (assigned to a)
>>=	a >>= b	a, right-shifted b bits (assigned to a)
<<=	a <<= b	a, left-shifted b bits (assigned to a)
&=	a &= b	a AND b (assigned to a)
=	a = b	a OR b (assigned to a)
^=	a ^= b	a XOR b (assigned to a)
,	e1,e2	e2 (e1 evaluated first)

These VAX C operators fall into the following categories:

- Unary operators, which take a single operand.
- Binary operators, which take two operands and perform a variety of arithmetic and logical operations.
- The ternary operator, which is the conditional operator, takes three operands and evaluates either the second or third expression, depending on the evaluation of the first expression.
- Assignment operators, which assign a value to a variable, optionally performing an additional operation before the assignment takes place.
- The comma operator, which guarantees left-to-right evaluation of comma-separated expressions.
- Primary operators, which usually modify or qualify identifiers (see Section 7.2 for more information).

Table 7-2 presents the precedence by which the compiler evaluates operations. Operators with the highest precedence appear at the top of the table; those with the lowest appear at the bottom. Operators of equal precedence appear in the same row.

Table 7-2: Precedence of VAX C Operators

Category	Operator	Associativity
Primary	() [] -> .	Left to right
Unary	! ~ ++ -- (type) * & sizeof	Right to left
Binary (mult.)	* / %	Left to right
Binary (add.)	+ -	Left to right
Binary (shift)	<< >>	Left to right
Binary (relat.)	< <= > >=	Left to right
Binary (equal.)	= = !=	Left to right
Binary (bitand)	&	Left to right
Binary (bitxor)	^	Left to right
Binary (bitor)		Left to right
Binary (AND)	&&	Left to right
Binary (OR)		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

Consider the following expression:

A*B+C

The identifiers A and B are multiplied first because the multiplication operator (*) has a higher precedence than the addition operator (+). The associative rule applies to each row of operators. Consider the following expression:

A/B/C

This expression is evaluated as follows because the division operator evaluates from left to right:

(A/B)/C

7.4 Unary Expressions and Operators

You form unary expressions by combining a unary operator with a single operand. All unary operators are of equal precedence and group from right to left. They perform the following operations:

- Negate a variable arithmetically (–) or logically (!) (Section 7.4.1)
- Increment (++) and decrement (– –) variables (Section 7.4.2)
- Find addresses (&) and dereference pointers (*) (Section 7.4.3)
- Calculate a one’s complement (~) (Section 7.4.4)
- Force the conversion of data from one type to another (the cast operator) (Section 7.4.5)
- Calculate the sizes of specific variables or of types (**sizeof**) (Section 7.4.6)

7.4.1 Negating Arithmetic and Logical Expressions

Consider the syntax of the following expression:

- expression

This is the arithmetic negative of expression. The compiler performs the arithmetic conversions. The negative of an **unsigned** quantity is computed by subtracting its value from 2^{32} . There is no unary plus operator in VAX C.

Consider the following expression:

! expression

The result is the logical (Boolean) negative of the expression. If the result of the expression is 0, the negated result is 1; if the result of the expression is not 0, the negated result is 0. The type of the result is **int**. The expression can be a pointer (or other address-valued expression) or an expression of any arithmetic type.

7.4.2 Incrementing and Decrementing Variables

Consider the syntax of the following expression:

++lvalue

The object to which the lvalue refers to in the expression is incremented before its value is used. After evaluating this expression, the result is the incremented rvalue, not the corresponding lvalue. For this reason, expressions that use the increment and decrement operators in this manner

cannot appear by themselves on the left side of an assignment expression where an lvalue is needed.

Consider the syntax of the following expression:

```
lvalue++
```

The object to which the lvalue refers to in the expression increments after its value is used. The expression evaluates to the value of the object *before* the increment, not the incremented variable's lvalue.

If the operand is a pointer, the address is incremented by the length of the addressed object, not by the integer value 1. If declared as an integer, the variable increases or decreases by the value 1.

The objects of the following lvalues point to other variables:

```
--lvalue
```

```
lvalue--
```

These pointers decrement not by the integer value 1, but by the size of the addressed object. The data type of the variable determines the amount of the increment or decrement. If declared as a pointer, the variable increments or decrements by the size of the addressed object's data type. For example, if declared as a pointer to an integer, the variable increments or decrements by the value 4. For example:

```
int *ip;
char *cp;
ip--;          /* Decrement by 4 */
--cp;         /* Decrement by 1 */
```

When using the increment and decrement operators, do not depend on the order of evaluation of expressions. Consider the following ambiguous expression:

```
k = x[j] + j++;
```

Is the value of variable *j* in *x[j]* evaluated before or after the increment occurs? Do not assume which expressions the compiler will evaluate first. To avoid ambiguity, increment the variable in a separate statement.

7.4.3 Computing Addresses and Dereferencing Pointers

Consider the syntax of the following expression:

```
& identifier
```

The expression results in the lvalue (address) of the identifier. The ampersand operator (&) may not be applied to **register** variables or to bit fields in structures or unions.

NOTE

In VAX C, the compiler changes any **register** variable to which the ampersand operator applies to an **auto** variable. If you do not use /STANDARD=PORTABLE, the compiler issues no warning message; if you do use /STANDARD=PORTABLE, the compiler issues an appropriate message.

In the special context of argument lists, you may apply the ampersand operator to constants. This use of the ampersand operator passes constants to user-defined functions that expect arguments to be passed by reference. This is a VAX C extension and is not portable. For more information about manipulating argument lists, see Section 5.1.2. For more information about the VAX Procedure Calling and Condition-Handling Standard, see Section 13.1.

Because function identifiers and unqualified array identifiers are lvalues, you cannot apply the ampersand operator to these identifiers. If you apply the address of an operator to function identifiers or to unqualified array identifiers, VAX C considers this to be a redundant use of the ampersand operator and generates the appropriate error message when the /STANDARD=PORTABLE qualifier is specified.

When an expression evaluates to an address, as in the following example, the address is used to indirectly access the object to which the address refers:

```
* pointer
```

An expression using the indirection operator (*) evaluates to the object pointed to by a pointer or by an address-valued expression.

7.4.4 Calculating a One's Complement

Consider the syntax of the following expression:

```
~ expression
```

The result is the one's complement of the evaluated expression; it converts each 1-bit into a 0-bit and vice versa. The expression must be integral (an integer or character). The compiler performs necessary arithmetic conversions.

7.4.5 Forcing Conversions to a Specific Type

The cast operator forces the conversion of its operand to a specified scalar data type. Structures and unions may not appear as a cast operator. The operator consists of a data-type name, in parentheses, which precedes the operand expression, as follows:

```
(type-name) expression
```

The resulting value of the expression converts to the named data type, just as if the expression were assigned to a variable of that type. If the operand is a variable, its value converts to the named type. The variable's contents do not change. The type name has the following formal syntax:

```
type-name ::=  
    type-specifier abstract-declarator
```

In simple cases, type-specifier is the keyword for a data type, such as **char** or **double**. The type-specifier may also be a structure specifier, union specifier, an **enum** specifier, or a **typedef** name.

An abstract-declarator in a parameter declaration is a declaration without an identifier or data-type keyword, as shown in the following form:

```
abstract-declarator ::=  
    empty  
    ( abstract-declarator )  
    * abstract-declarator  
    abstract-declarator ( )  
    abstract-declarator [ constant-expression ]
```

Consider the following form of the abstract-declarator:

```
abstract-declarator( )
```

To avoid confusion with the previous form, the abstract-declarator may not be empty in the following form:

```
(abstract-declarator)
```

Abstract declarators may include the brackets and parentheses that indicate arrays and function calls. However, cast operations may not force the conversion of any expression to an array, function, structure, or union. The brackets and parentheses are used in operations such as the following example, which casts identifier P1 to “pointer to array of **int**.”

```
(int (*)[]) P1
```

This kind of cast operation does not change the contents of P1; it only causes the compiler to treat the value of P1 as a pointer to such an array. For

example, casting pointers this way can change the scaling that occurs when you add an integer to a pointer. For example:

```
int *ip;
((char*)ip)++; /* Increments by 1 not by 4 */
```

7.4.6 Calculating Sizes of Variables and Data Types

Consider the syntax of the following expressions:

```
sizeof expression
sizeof ( type-name )
```

The result is the size, in bytes, of the operand. In the first case, the result of **sizeof** is the size determined by the type of the expression. In the second case, the result is the size, in bytes, of an object of the named type. The syntax of type-name is the same as that for the cast operator. For example:

```
int x;
x = sizeof(char *);
```

See Section 7.4.5 for more information about the cast operator.

7.5 Binary Expressions and Operators

The binary operators are categorized as follows:

- Additive operators: addition (+) and subtraction (–) (Section 7.5.1)
- Multiplication operators: multiplication (*), mod (%), and division (/) (Section 7.5.2)
- Equality operators: equality (==) and inequality (!=) (Section 7.5.3)
- Relational operators: less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=) (Section 7.5.4)
- Bitwise operators: AND (&), OR (|), and XOR (^) (Section 7.5.5)
- Logical operators: AND (&&) and OR (||) (Section 7.5.6)
- Shift operators: left shift (<<) and right shift (>>) (Section 7.5.7)

The following sections describe these binary expressions and operators.

7.5.1 Additive Operators

The additive operators (+) and (−) perform addition and subtraction. Their operands are converted, if necessary, following the arithmetic conversion rules. For more information, see Section 7.9.1.

You can increment an array pointer by adding an integral variable to the address of an array element. The compiler calculates the size of one array element, multiplies that by the integer to obtain the offset value, and then adds the offset value to the address of the designated element. For example:

```
int arr[10];
int *p = arr;
p = p + 1; /* Increments by 4 */
```

You may subtract a value of any integral type from a pointer or address; in that case, the same conversions apply as for addition.

When you add or subtract two **enum** constants or variables, the type of the result is **int**.

If you subtract two addresses of objects of the same type, the result converts (divides by the length of the object) to an **int** representing the number of objects separating the addressed objects. The result of this conversion is unpredictable unless the two objects are in the same array.

7.5.2 Multiplication Operators

The multiplication operators (*), (/), and (%) perform arithmetic conversions, if necessary. The binary operator (*) performs multiplication. The binary operator (/) performs division. When integers are divided, truncation is toward zero.

The binary mod operator (%) divides the first operand by the second and yields the remainder. Both operands must be integral. The sign of the result is the same as the sign of the quotient. If variable b is not zero, then the following statement is true:

```
(a/b)*b + a%b = a
```

7.5.3 Equality Operators

The equality operators equal-to (==) and not-equal-to (!=) perform the necessary arithmetic conversions on their two operands. These operators produce a result of type **int**, so that in the following statement the result is the value 1, if both relational expressions have the same truth value, and the value 0 if they do not:

```
a<b == c<d
```

Two pointers or addresses are equal if they identify the same storage location. You can compare a pointer or address with an integer, but the result is not portable unless the integer is zero; a null pointer is considered equal to zero.

Although different symbols are used for assignment and equality, (=) and (==) respectively, VAX C allows either operator in some contexts, so you must be careful not to confuse them. Consider the following example:

```
if (x=1) statement-1;
else     statement-2;
```

In the previous example, statement-1 always executes, since the result of assignment x=1 delimited by parentheses is equivalent to the value of x, which is equal to 1 (or true).

NOTE

The following example shows a coding practice useful to avoid this common error when doing comparisons. By placing the constant first, the compiler diagnoses the incorrect use of the equality operator (=).

```
int x;
if (1==x); /* This syntax does the comparison */
if (1=x);  /* This syntax causes a compiler error */
```

7.5.4 Relational Operators

The relational operators compare two operands and produce a result of type **int**. The result is the value 0 if the relation is false, and 1 if it is true. The operators are less-than (<), greater-than (>), less-than or equal-to (<=), and greater-than or equal-to (>=). The compiler performs necessary arithmetic conversions.

If you compare two pointers or addresses, the result depends on the relative locations of the two addressed objects. Pointers to objects at lower addresses are less than pointers to objects at higher addresses. If two addresses indicate elements in the same array, the address of an element with a lower subscript is less than the address of an element with a higher subscript.

The operators group from left to right. However, note that the following statement compares the variable `c` with 0 or 1 (possible results of `a<b`); it does not mean “if `b` is between `a` and `c` . . . ”:

```
if (a<b<c) . . .
```

In order to check that `b` is between `a` and `c`, you should use the following code:

```
if (a<b && b<c) . . .
```

7.5.5 Bitwise Operators

The bitwise operators may be used only with integral operands: with variables of types `char` and with `int` of all sizes. The compiler performs the necessary arithmetic conversions. The result of the expression is the bitwise AND (`&`), XOR—exclusive OR (`^`), or OR (`|`) of the two operands. The compiler evaluates all operands. Figure 7–1 shows the effects of Boolean algebra when using the bitwise operators.

In Boolean algebra, VAX C evaluates values bit by bit. If you are using the bitwise AND on a bit value 1 and on a bit value 0, the result is 0. When using the bitwise AND, both bits must be 1, as shown in Figure 7–1, for the result to be 1. When using the bitwise OR, either bit value can be 1 for the result to be 1. When using the bitwise EXCLUSIVE-OR, either value, but not both, can be 1 for the result to be 1.

7.5.6 Logical Operators

The logical operators are AND (`&&`) and OR (`||`). These operators guarantee left-to-right evaluation. The result of the expression (of type `int`) is either 0 (false) or 1 (true). If the compiler can make an evaluation by examining only the left operand, it does not evaluate the right operand. Consider the following expression:

Figure 7-1: Boolean Algebra and the Bitwise Operators

Boolean Algebra

AND (&)

	1	0
1	1	0
0	0	0

OR (|)

	1	0
1	1	1
0	1	0

EXCLUSIVE-OR (^)

	1	0
1	0	1
0	1	0

OPERATOR	BITWISE OPERATION							DECIMAL VALUE
AND (&)	1	0	1	1	1	1	1	95
	1	1	0	0	0	0	1	97
	1	1	0	0	0	0	1	65
OR ()	1	0	1	1	1	1	1	95
	1	1	0	0	0	0	1	97
	1	1	1	1	1	1	1	127
X-OR (^)	1	0	1	1	1	1	1	95
	1	1	0	0	0	0	1	97
	0	1	1	1	1	1	0	62

ZK-3071-GE

E1 && E2

The result is 1 if both its operands are nonzero, or 0 if one operand is 0. If expression E1 is 0, expression E2 is not evaluated. Similarly, the following expression is 1 if either operand is nonzero, and 0 otherwise. If expression E1 is nonzero, expression E2 is not evaluated.

E1 || E2

The operands of logical operators need not have the same type, but each must be one of the fundamental types or must be a pointer or other address-valued expression.

7.5.7 Shift Operators

The shift operators (`<<`) and (`>>`) take two operands, both of which must be integral. The compiler performs necessary arithmetic conversions on both operands if they are not integers. The right operand is then converted to **int**, and the type of the result is the type of the left operand. Consider the result of the following expression:

```
E1 << E2
```

The result is the value of expression `E1` shifted to the left by `E2` bits. The compiler clears vacated bits. Consider the following expression:

```
E1 >> E2
```

The result is the value of expression `E1` shifted to the right by `E2` bits. The compiler clears vacated bits if `E1` is **unsigned**; otherwise, the bits are filled with a copy of `E1`'s sign bit.

The result of the shift operation is undefined if the right operand (`E2` in the previous example) is negative or if the value of `E2` is greater than 32.

7.6 Conditional Operator

The conditional operator (`?:`) takes three operands. It tests the result of the first operand and then evaluates one of the other two operands based on the result of the first. Consider the following example:

```
E1 ? E2 : E3
```

If expression `E1` is nonzero (true), then `E2` is evaluated. If `E1` is 0 (false), `E3` is evaluated. Conditional expressions group from right to left. The compiler makes conversions in the following order:

1. If possible, the arithmetic conversions are performed on expressions `E2` and `E3`, so that they will result in the same type.
2. If expressions `E2` and `E3` are address expressions indicating objects of the same type, the result has that type.
3. One of the `E2` and `E3` operands must be an address expression, and the other, the constant 0. The result has the type of the addressed object.

7.7 Assignment Expressions and Operators

VAX C has several assignment operators. An assignment is not only an operation but is also an expression. Assignments result in the value of the target variable after the assignment. They can be used as subexpressions in larger expressions.

The set of assignment operators consists of the equal sign (=) alone and in combination with binary operators. An assignment expression has two operands (an lvalue and an expression separated by one of these operators). Consider the following assignment expression:

```
E1 += E2;
```

This is equivalent to the following expression:

```
E1 = E1 + E2;
```

The expression E1 is evaluated once and must result in an lvalue. The type of the assignment expression is the type of E1, and the result is the value of E1 after the completion of the operation. You must delimit some expressions in parentheses if the expressions may contain other operators of a lower precedence. Consider the following expression:

```
a *= b + 1;
```

This is the same as the following expression:

```
a = a * (b + 1);
```

However, the previous expression is not the same as the following expression:

```
a = (a * b) + 1;
```

In the following assignment expression, the value of expression E2 replaces the previous object of E1:

```
E1 = E2
```

The following expression adds 100 to the contents of `a_number[1]`:

```
a_number[1] += 100;
```

The result of this expression is the result of the addition and has the same type as `a_number[1]`.

If both assignment operands are arithmetic, the right operand is converted to the type of the left before the assignment (see Section 7.9.1).

You can use the assignment operator (=) to assign values to structure and union members. You can assign one structure value to another as long as you define the structures to be the same size. With all other assignment operators, all right operands and all left operands must be either pointers or evaluate to arithmetic values. If the operator is (-=) or (+=), the left operand may be a pointer, and the right operand (which must be integral) is converted in the same manner as the right operand in the binary plus (+) and minus (-) operations.

You can assign an address to an integer, an integer to a pointer, and the address of an object of one type to a pointer of another type. Such assignments are simple copy operations, with no conversions. This usage may cause addressing exceptions when you use the resulting pointers. However, if the constant 0 is assigned to a pointer, the result is a null pointer. The equality operators distinguish a null pointer from a pointer that points to any object.

For compatibility with some other C implementations, VAX C allows certain deviations from the spellings of the compound assignment operators shown in Table 7-2. The deviations are as follows:

- When the operators are written in the order shown in Table 7-2, the two characters can be separated by blank spaces. For example, the following expressions are identical:

```
E1 += E2;  
E1 + = E2;
```

- The operators can also be written with the characters in reverse order, as in the following expression:

```
E1 =+ E2;
```

The second form generates an informational message. Avoid this form for the following reasons:

- The syntax allowed by VAX C is more restrictive in this case. Specifically, the characters (*, -, and &) must be immediately adjacent to the equal sign (=) character because they also appear in unary operators. For example, this placement avoids ambiguities such as that shown in the following example, which multiplies the result of expression E1 by the value of p:

```
E1 =*p;
```

- Even with usage that follows the guidelines, it is possible to introduce ambiguities, as in the following expression:

```
E1 = /*part of a comment . . .
```

7.8 Comma Expression and Operator

When two expressions are separated by the comma operator, they evaluate from left to right, and the compiler discards the result of the left expression. If you separate many expressions with commas, the compiler discards all but the result of the rightmost expression. For example, the following expression assigns the value 1 to variable R and the value 2 to variable T:

```
R = T = 1,    T += 2,    T -= 1;
```

The type and value of the result of a comma expression are the type and value of the rightmost operand. The operator evaluates operands from left to right.

You must delimit comma expressions with parentheses if they appear where commas have some other meaning, as in argument and initializing lists. Consider the following expression:

```
f(a, (t=3,t+2), c)
```

This example calls the function, f, with the arguments a, 5, and c. In addition, variable t is assigned the value 3.

7.9 Data-Type Conversions

VAX C performs data-type conversions in the following four situations:

- When two or more operands of different types appear in an expression (including an assignment).
- When arguments other than long integers, addresses, or double-precision, floating-point numbers are passed to a function.
- When arguments that do not conform exactly to the parameters declared in a function prototype are passed to a function.
- When the data type of an operand is deliberately converted by the cast operator. See Section 7.4.5 for more information on the cast operator.

The following sections describe how to convert operands and function arguments.

7.9.1 Converting Operands

The following rules—referred to as the *arithmetic conversion rules*—govern the conversion of operands in arithmetic expressions. Although they do not specify explicit conversions at the machine-language level, the rules govern in the following order:

1. Any operands of type **char** or **short** (signed or unsigned) convert to their 32-bit equivalents (**int** or **unsigned int**). Any operands of type **float** convert to **double** unless `/PRECISION=SINGLE` is specified.
2. If either operand is **double**, the other converts to **double**, and that is the type of the result.
3. If either operand is **unsigned**, the other converts to **unsigned**, and that is the type of the result.
4. Both operands must be **int**, and that is the type of the result.

The arithmetic conversions are performed on all arithmetic operands. Some operators, such as the shift operators (`>>`) and (`<<`), require integers as operands. If one operand is of type **float** or **double**, you cannot meet this requirement.

In previous versions of VAX C, floating-point arithmetic was carried out in double precision. Since the proposed ANSI C standard no longer requires this conversion, VAX C attempts to perform arithmetic in single precision if `/PRECISION=SINGLE` is specified on the compilation. If an operand of type **float** appears in an expression, it is treated as a single-precision object unless the expression also involves an object of type **double**, in which case the usual arithmetic conversion applies.

When an operand of type **double** is converted to **float**, (for example, by an assignment) the compiler rounds the operand before truncating it to **float**.

The compiler may convert a **float** or **double** value operand to an integer by assignment to an integral variable. In VAX C, the truncation of the **float** or **double** value is always toward zero.

Conversions also take place between the various kinds of integers. In VAX C, variables of type **char** are bytes treated as signed integers. When a longer integer is converted to a shorter integer or to **char**, it is truncated on the left; excess bits are discarded. For example:


```
int i;
char c;

i = 0xFFFFFFFF41;
c = i;
```

This code assigns hex 41 ('A') to variable c. The compiler converts shorter signed integers to longer ones by sign extension.

Whenever the compiler combines an unsigned integer and a signed integer, the signed integer converts to **unsigned** and the result is **unsigned**. All conversions from signed to unsigned perform an intermediate conversion to **int**. For example, the compiler converts a **char** or **short** operand to an unsigned version by first converting it to a signed **int** and then by truncating it to form the unsigned version. All conversions from unsigned to signed (such as conversions done with the cast operator) involve an intermediate conversion to **unsigned int**.

You can also add integers to pointers, in which case the integer is scaled (multiplied) by a factor that depends on the type of the object to which the pointer points. See Section 7.5.1 for more information about scaling pointers.

7.9.2 Converting Function Arguments

The data types of function arguments are assumed to match the types of the formal parameters unless a function prototype declaration is present. In the presence of a function prototype, all arguments in the function invocation are compared for assignment compatibility to all parameters declared in the function prototype declaration. If the type of the argument does not match the type of the parameter but is assignment compatible, VAX C converts the argument to the type of the parameter (see Section 7.9.1). If an argument in the function invocation is not assignment compatible to a parameter declared in the function prototype declaration, VAX C generates an error message.

Unless a function prototype is present, all arguments of type **float** convert to **double**, all variables of type **char** and **short** convert to **int**, all variables of type **unsigned char** and **unsigned short** convert to **unsigned int**, and an array or function name converts to the address of the named array or function. The compiler performs no other conversions automatically, and any mismatches after these conversions are programming errors.

Use the cast operator to pass arguments to parameters of different types. See Section 7.4.5 for more information on the cast operator. For more information about manipulating argument lists, see Section 5.1.2. For more information about the VAX Procedure Calling and Condition-Handling Standard, see Section 13.1.

Data Types and Declarations

The values of both constants and variables have *data types*. Data types specify the amount of storage required and how to interpret the data object in that storage space. This chapter discusses the following topics in respect to data types:

- Constants (Section 8.1)
- Variables (Section 8.2)
- Integers (Section 8.3)
- Character constants (Section 8.3.2)
- Floating-point numbers (Section 8.4)
- Pointers (Section 8.5)
- Enumerated types (Section 8.6)
- Arrays (Section 8.7)
- Character-string variables (Section 8.8)
- Structures and unions (Section 8.9)
- The **void** keyword (Section 8.10)
- The **typedef** keyword (Section 8.11)
- Interpreting declarations (Section 8.12)

8.1 Constants

You can represent data in VAX C using constants. A *constant* is a primary expression with a defined value that does not change. You may represent a constant in a literal form, which contains the explicit numbers, letters, and operators that comprise the constant, or, you may define a symbol to represent the constant value. (For more information about symbolic representation of constants, see Section 10.1.) Constants, like all data in

VAX C, have data types. The data type determines the amount of storage needed and determines how to interpret the stored object or constant value. The compiler determines the data type of constants by the way in which their values are represented in the source code.

8.2 Variables

You can also represent data in VAX C using variables, whose values can change throughout the execution of the program. All variables used in a program must be declared. When you declare a variable, you specify the data type of the stored object. An *object*, in VAX C, is a value requiring storage. Declarations determine the size of a storage allocation; definitions initiate the allocation of storage. See Section 8.2.1 for more information about the data types of variables.

Unlike constants, variables can be declared and defined. Most variable declarations are also definitions because storage is allocated at that point in the program. To declare a variable, specify the data type. To define a variable, assign the variable the proper storage class and place the variable declaration within the program structure. Also, if you can initialize a variable in the declaration, the variable is defined. For more information about variable definitions, scope, and storage allocation, see Chapter 9.

8.2.1 Classification of Variables

There are two kinds of variables: *scalar* and *aggregate*. Scalar variables have objects that can be manipulated arithmetically in their entirety. These objects are single characters, individual numbers, and pointers. Aggregate variables are data structures (arrays, structures, and unions) that are comprised of distinct elements (members) that you can declare to be of either a scalar or aggregate data type.

VAX C has defined data-type keywords for your use in declaring program variables. The following sections describe the VAX C data-type keywords and the format for a VAX C variable declaration.

8.2.1.1 Data-Type Keywords

To declare or define variables, you need to know the VAX C keywords associated with each data type. Table 8–1 lists the VAX C data-type keywords according to classification.

Table 8–1: VAX C Data-Type Keywords

Scalar Keywords	Aggregate Keywords	Other Type Keywords
int	struct	void
long	union	
unsigned	variant_struct	
short	variant_union	
char		
float		
double		
enum		

In the sections that follow, the keywords and operators used to declare variables of given data types are listed in the section header for ease of reference.

VAX C also supports the type modifiers **const** and **volatile**. For information about these type modifiers, see Section 9.7.

8.2.1.2 Format of a Variable Declaration

A variable declaration can be composed of the following items:

- Data-type specifiers such as a data type or data-type modifier keyword, one structure, union, or **enum** tag, and if necessary, a **typedef** name
Any of these give the data type of the declared object.
- An optional storage-class keyword
A storage-class keyword affects the scope of a variable and determines how it is stored. If you omit the storage-class keyword, there is a default storage class that depends upon the physical location of the declaration in the program. The positions of the storage-class keywords and the data-type keywords are interchangeable.

- Declarators, which list the identifiers of the declared objects and which may contain operators that declare a pointer, function, or array of objects of the declared type
- Initializers for each declared object or aggregate element giving the initial value of a scalar variable or the initial values of structure members or array elements

An initializer consists of an equal sign (=) followed by either a single expression or a comma-list of one or more expressions in braces.

Consider the following example:

```
int var_number = 10;
```

This declaration both declares and defines the integer variable, `var_number`, that has an initial value of 10. The keyword `int` specifies the amount of storage needed on a VAX system for an integer. The identifier `var_number` follows. The equality operator (=) initializes the variable with the literal constant 10; for the initialization to take place, storage is allocated and the variable is defined. Declarations must end in a semicolon (;).

The variable declaration in the previous example is not difficult to interpret, but even experienced VAX C programmers have difficulty interpreting complex variable declarations. See Section 8.12 for more information about interpreting VAX C variable declarations.

8.3 Integers (int, long, short, char, and unsigned)

Integer variables are declared with the keywords `int`, `long`, `short`, `char`, and `unsigned`. The following is an example of an integer declaration:

```
int x;
```

Character variables are declared with the keyword `char`. An example of a character declaration with the initialization of a character variable is as follows:

```
char ch = 'a';
```

Table 8–2 specifies the sizes and ranges of integers.

Table 8-2: Size and Range of VAX C Integers

Keyword	Size	Range
int, long, and long int	32 bits	-2,147,483,648 to 2,147,483,647
unsigned and unsigned int	32 bits	0 to 4,294,967,295
short and short int	16 bits	-32,768 to 32,767
unsigned short	16 bits	0 to 65,535
char	8 bits	-128 to 127
unsigned char	8 bits	0 to 255

The following sections describe the constants that you can assign to the integer variables.

8.3.1 Integer Constants

There are three types of integer constants: decimal, hexadecimal, and octal; these constants can be signed or unsigned. Integer constants can consist of the characters 0 to 9, a to f (for hexadecimal integers), A to F (also for hexadecimal integers), and, optionally, the characters x, X, l, L, u and U, in either upper- or lowercase letters. Use the characters x and X to specify hexadecimal numbers. The characters l and L specify that the constant is to be considered as a **long** integer (4 bytes, 1 longword). The characters u and U specify that the integer constant is unsigned rather than signed. On other implementations of the C language, values of the **int** data type may require only 16 bits of storage. On a machine with the VAX architecture, values of the **int** data type require 32 bits of storage. Therefore, note that values of the **int** and **long** data types require identical storage on a machine with the VAX architecture. VAX C supports the L suffix only for the sake of program portability.

You can specify integer constants in decimal, octal, and hexadecimal radices. An integer constant is assumed to be decimal unless it begins with 0 or 0x; if it begins with 0, it is assumed to be octal; if it begins with 0x, it is assumed to be hexadecimal.

In octal constants, the digits 8 and 9 have the octal values 010 and 011, respectively. For instance, the octal number 039 is equal to $3 * 8 + 9$, or decimal value 33; the octal number 080 is equal to $8 * 8 + 0$, or decimal value 64.

Even though VAX C supports the digits 8 and 9 in octal constants, you should avoid using these octal constants so as not to conflict with other implementations of the C language.

Integer constants must not include a decimal point; constants with a decimal point are of type **double**. Integer constants that exceed a longword are treated as programming errors.

Character constants such as 'a' and '\$' are also valid integer constants. Their integer values in VAX C are the values of the corresponding ASCII codes.

Some examples of valid integer constants are as follows:

```
133L           /* Long decimal integer           */
0x17A         /* Hexadecimal integer           */
056           /* Octal integer                 */
4294967295ul  /* Unsigned long integer         */
077u         /* Unsigned octal integer        */
'a'          /* Decimal 97                    */
'$'          /* Decimal 36                    */
```

Some examples of invalid integer constants are as follows:

```
143.         /* Includes a decimal point      */
3333333333   /* Out of range for int          */
+333333      /* '+' is an invalid character   */
77af         /* Hexadecimal constants must be
 * prefixed with "0x"           */
```

8.3.2 Character Constants

A *character constant* is a value, requiring at least 8 bits (1 byte) or at most 32 bits (1 longword) of memory, that is enclosed in apostrophes. Character constants can be a single ASCII character, as in the following example:

```
char ch = 'a'; /* Lowercase letter 'a' is a constant
 * assigned to ch. */
```

The character constant 'a' has the ASCII value of 97. If the value of a character constant is not large enough to fill 32 bits of memory, the compiler stores the character or characters in the low-order byte(s) and pads the remaining bytes with NUL characters ('\0').

Character constants do not have to be single characters, as shown in the following example (please note that this is VAX C specific and not portable):

```
int l_word = 'a:cd' /* This constant contains 4 characters */
printf("%c\n", l_word);
printf("%.4s", &l_word); /* String with maximum 4 characters */
```

Sample output from this program is as follows:

```
$ RUN EXAMPLE
a
a:cd
$
```

If you print variable `l_word` as a character, the `printf` function prints only the character located in the low-order byte of the integer allocation. To print all of the characters in the longword allocated to the variable, you have to print the variable as a string and pass the address of the integer variable as an argument. If you print the integer variable as a string, be sure to specify a precision of at most 4, since you can never be sure if the next byte in the string is a terminating NULL character.

The apostrophe (`'`) and quotation mark (`"`) are significantly different punctuation marks in VAX C, indicating a character constant and a string constant, respectively. One context in which the difference is important is in an argument list. If you specify a function argument as a string, and wish to pass a character constant, you must enclose the character in quotation marks, not apostrophes, even if the string is only one to four characters in length. See Section 8.8.1 for more information about character-string constants.

8.3.3 Escape Sequences

In VAX C, escape sequences are character strings that represent a single printing or nonprinting character. The term “escape sequences” does not designate a string beginning with the ASCII character ESC, as in VT100 escape sequences.

Table 8–3 presents the escape sequences that specify the nonprinting characters, the apostrophe, and the backslash (`\`).

Table 8–3: VAX C Escape Sequences

Character	Mnemonic	Escape Sequence
newline	NL	<code>\n</code>
horizontal tab	HT	<code>\t</code>
vertical tab	VT	<code>\v</code>
backspace	BS	<code>\b</code>
carriage return	CR	<code>\r</code>
form feed	FF	<code>\f</code>
backslash	<code>\</code>	<code>\\</code>
apostrophe	<code>'</code>	<code>\'</code>
quotes	<code>"</code>	<code>\"</code>
bit pattern	ddd	<code>\ddd</code> or <code>\xddd</code>

An escape sequence, such as `'\n'`, denotes a single character.

The form `'\ddd'` is used to specify any byte value (usually an ASCII code), where the digits `ddd` are one to three octal digits. The octal digits are limited to 0 through 7. A common use is to specify the ASCII NUL character, as follows:

```
'\0'
```

Similarly, the form `'\xddd'` is used to specify any byte value (usually an ASCII code), where the digits `ddd` are used to specify one to three hexadecimal digits.

The following are examples of valid escape sequences of the form `'\ddd'` and `'\xddd'`. Both of these escape sequences are used to specify an a-umlaut (ä) on a VT200-series terminal in octal and hexadecimal digits, respectively.

```
'\344'  
'\xe4'
```

If the character following the backslash in an escape sequence is illegal, the backslash is ignored; that is, the character constant's value is the same as if the backslash were not present.

8.4 Floating-Point Numbers (float and double)

When declaring floating-point variables, you determine the amount of precision needed for the stored object. In VAX C, you can have either single-precision or double-precision variables. If you choose double precision, you have the choice of using either the `D_floating` or `G_floating` formats.

The sizes and ranges of VAX C floating-point numbers are as follows:

- **float**

Float is a 32-bit keyword with a range of:

$$0.29 * 10^{-38} \text{ to } 1.7 * 10^{38}$$

These values are precise to 7 decimal digits.

- **double D_floating**

Double `D_floating` is a 64-bit keyword with a range of:

$$0.29 * 10^{-38} \text{ to } 1.7 * 10^{38}$$

These values are precise to 16 decimal digits.

- **double G_floating**

Double `G_floating` is a 64-bit keyword with a range of:

$$0.56 * 10^{-308} \text{ to } 0.899 * 10^{308}$$

These values are precise to 15 decimal digits.

You use the keyword **float** to declare a single-precision, floating-point variable, represented internally in the VAX `F_floating-point`, binary format.

The keyword **double** declares a double-precision, floating-point variable. You can use the keywords **double** and **long float** interchangeably. However, **long float** should not be used so as to avoid conflict with other implementations of the C language. There are two representations of the VAX C data type **double**: `D_floating` and `G_floating`.

The `G_floating` precision, approximately 15 digits, is less than that of variables represented in `D_floating` format. The fractional portion of the variable may contain one more digit, but the integral portion of the variable must contain one less digit.

The default representation of the data type **double** is `D_floating`. The `G_floating` representation is chosen by compiling the program with the `/G_FLOAT` qualifier on the DCL command line. For more information about the compilation command line, see Section 1.3.1. Modules compiled with the `D_floating` representation should not be linked with modules compiled with the `G_floating` representation. Since there are no functions in the VAX C Run-Time Library (RTL) that will perform type conversions on files, use the VMS Run-Time Library (RTL) functions `MTH$CVT_D_G`, `MTH$CVT_G_D`, `MTH$CVT_DA_GA`, and `MTH$CVT_GA_DA` if you do not wish to recompile the program. For more information about using the VMS RTL, see the *VMS Run-Time Library Routines Volume*

NOTE

Modules must be linked to the appropriate run-time library. For more information about linking against object libraries, see Section 1.4.5.2. For more information about linking against shareable images, see Section 1.4.5.3.

8.4.1 Floating-Point Constants

A floating-point constant has an integral part, a decimal point, a fractional part, the letter `e` or `E`, and an optionally signed integer exponent. The integral and fractional parts consist of decimal digits; you may omit either the integral or fractional part. You may omit either the decimal point with the following digits or the exponent(`e,E`), but not both.

All floating-point constants are of type **double**.

The following are examples of floating-point constants:

```
3.0e10
3.0E-10
3.0e+10
3E10
3.0
.120e2
.120
```

8.5 Pointers

Pointers in VAX C are variables that contain 32-bit addresses of other objects. They are declared with the asterisk operator and the data type of the object to which it points, as in the following example:

```
int *px;
```

Identifier `px` is declared as a pointer to a variable of type **int**; the construct `*px` is treated as a variable of type **int**. An expression such as `*px` yields the object to which `px` points.

The unary asterisk (`*`) is also the indirection operator in VAX C. The unary asterisk operates as follows:

```
x = *px;
```

This statement assigns the value of the object pointed to by `px` to variable `x`. Since the asterisk can be used in any sort of declarator, you can have pointers to scalars, to functions, to other pointers, to structures, and so forth.

The following operations are legal using pointer variables:

- Assigning pointers of the same type to each other
- Adding or subtracting a pointer and an integer
- Subtracting or comparing two pointers to members of the same array
- Assigning or comparing to zero
- Assigning a pointer of any type to a **void** pointer (see Section 8.5.1)
- Assigning **void** pointers to a pointer of any type (see Section 8.5.1)

All other pointer arithmetic is illegal.

Static and extern pointers are null unless initialized. A null pointer is a pointer variable that has been assigned the integer constant 0. An auto pointer that is not initialized contains an unknown value.

If you try to access data by means of a null pointer, the VMS system returns the hardware error, ACCVIO. The address space between value 0 and 511 (decimal value, 1 page) is not accessible because it is not mapped into the program's virtual address space. This is true for all VAX C programs.

If you include either of the files `stdio.h` or `stddef.h` in your program, you can compare the value of a pointer variable to the predefined macro `NULL` to see if it is a null pointer. For more information about these include files, see the *VAX C Run-Time Library Reference Manual*. For more information on how to include files into your program, see Section 10.4.

When used in certain arithmetic expressions, the compiler uses the size of the object of the pointer. For example, if `px` is a pointer to an integer, `px + 1` evaluates to the next integer address, 4 bytes after `px`. If `px` is a pointer to **char**, `px + 1` yields the next character address, 1 byte after `px`. The compiler uses the type of the pointer's object to scale the arithmetic.

A different result occurs with an expression such as the following:

```
*px + 1
```

This expression evaluates to the value of the object to which `px` points added to 1.

Some contexts may require a pointer of a particular type. This is necessary, for example, if a function requires that an argument be passed by reference.

The ampersand (&) operator is used to take the address. Consider the following example:

```
px = &x;
```

This statement assigns the address of variable `x` to pointer `px`. After an assignment such as this, a reference to `*px` yields the value of `x`.

You should not apply the ampersand operator to constants, to **register** variables, to function identifiers, or to array identifiers. VAX C allows the application of the ampersand operator to constants so that you can pass constants, as arguments, to system service routines. For more information about instances where you would apply the ampersand operator to a constant, see Section 13.2.

The compiler stores constant values in a read-only program section (psect), so attempts to change the value by applying the ampersand operator will result in an error. For more information about psects, see Chapter 14. For information about the relationship between VAX C storage classes and psects, see Section 9.2.

If you do apply the ampersand to **register** variables, the compiler changes the variable's storage class from **register** to **auto**. If you use the `/STANDARD=PORTABLE` qualifier, the compiler generates an informational message; if you do not use this qualifier, the compiler changes the storage class without notification.

If you apply the ampersand operator to function or array identifiers, VAX C issues a message, since asking for the address of an expression returning an address is redundant.

8.5.1 void Pointers

The **void** pointer is a pointer that does not have a specified data type to describe the object to which it points. In effect, this is a generic pointer. (In the past, VAX C programmers have used **char** * to define generic pointers; this practice is now discouraged for portability reasons.)

You can assign a pointer of any type to a **void** pointer without a cast (see Section 7.4.5 for more information on the cast operation). For example, you can use this type of pointer in function calls, in function arguments, or in function prototypes when the parameter or return value is a pointer of an unknown type. Consider the following example:

```
main()
{
    void *generic_pointer;
    .
    .
    .
    /* If the function return value can be a pointer to many types . . . */
    generic_pointer = func_returning_pointer( arg1, arg2, arg3 );
    .
    .
}
```

The following statements are also valid:

```
main()
{
    float *float_pointer;
    void *void_pointer;
    .
    .
    .
    float_pointer = void_pointer;
    /* Or . . . */
    void_pointer = float_pointer;
    .
    .
}
```

See Section 5.1.2 for information about using **void** in function definitions.

8.6 Enumerated Types (enum)

An *enumerated type* is a user-defined data type that is not derived from other fundamental types. Each listed enumerator is associated with an incremented integer constant starting with zero. The following example shows the declaration of a variable and an enumeration type, or tag:


```
enum shades
{
    out, verydim, dim, prettybright, bright
} light;
```

This declaration defines the variable `light` to be of an enumerated type `shades`. The variable can assume any of the enumerated values.

The tag `shades` becomes the enumeration tag of the new type; `out`, `verydim`, `...`, `bright` are the enumerators with values 0 through 4. These enumerators are the constant values of the type `shades` and can be used wherever integer constants are valid.

If the tag has been declared, you can use the tag as a reference to that enumerated type, as in the following declaration:

```
enum shades light1;
```

The variable `light1` is an object of the enumerated data type, `shades`.

An **enum** tag can have the same spelling as other identifiers in the same program, including variable identifiers and member names in structures and unions, because the meanings are distinguished by context. However, **enum** constant names must be spelled uniquely. VAX C allows forward referencing to **enum** tags that have not been declared yet in the source code, but are declared further on in the program.

Internally, each enumerator is associated with an integer constant; the compiler gives the first enumerator the value 0 by default, and the remaining enumerators are incremented by the value 1, as they are read from left to right. Any enumerator can be set to a specific integer constant value. The enumerators to the right of such a construct (unless they are also set to specific values) then receive values that are 1 greater than the previous value. Consider the following example:

```
enum spectrum
{
    red, yellow=4, green, blue, indigo, violet
} color2;
```

This declaration gives `red`, `yellow`, `green`, `blue`, `...`, the values 0, 4, 5, 6, `...`

Examining the value of a variable like `color2` displays an integer, not a string such as `red` or `yellow`. Although they are stored internally as integers, regard enumerated data types as distinct from the fundamental types.

Type mismatches between the enumerated and fundamental types, or between different enumerated types, are errors. The **enum** types in the following example are not valid:

```

enum
{
    red, orange, yellow, green, blue, indigo, violet
} color1;
enum illum
{
    out, verydim, dim, prettybright, bright
} light;
light = red;

```

The enumerators `red` and `light` have different enumerated types.

The **enum** type in the following example is also not valid:

```

enum illum
{
    out, verydim, dim, prettybright, bright
} light;
light = 1;

```

Value `1` is not an enumerated value for variable `light`.

To perform valid mixed-type operations, use the cast operator. Consider the following example:

```

/* Both evaluate to verydim (1) */
light = (enum illum) (out + (enum illum) red);
light = (enum illum) 1;

```

Here, the cast operation `(enum illum)` causes the compiler to treat **enum** constant `red` and integer constant `1` as values of enumerated type `illum`.

Variables and enumerators of enumerated types take on various storage classifications when used with the **globaldef** and **globalref** storage-class keywords. For more information about the use of these storage-class keywords with enumerated types, see Section 9.6.3.

8.7 Arrays ([])

Arrays are declared with the square bracket operator (`[]`), as in the following declaration of a 10-element array of integers called `table_one`:

```
int table_one[10];
```

The type specifier **int** gives the data type of the elements. The elements of an array can be of any scalar or aggregate data type. The identifier `table_one` specifies the name of the array. The constant expression gives the number of elements in a single dimension. Array subscripts in VAX C begin with the integer `0` (not `1`); they must be integral. In the previous example,

the first element is `table_one[0]` and the last element is `table_one[9]`. Unpredictable results may occur if you specify a subscript larger than or equal to the declared dimension bound; you would then be accessing objects outside of the memory allocated to the array. The use of array subscripts in the following example is not recommended:

```
int table_one[10];
table_one[10] = 69;
table_one[5] = table_one[11];
```

VAX C supports multidimensional arrays: arrays declared as an array of arrays. Consider the following example:

```
int table_one[10][2];
```

Here, variable `table_one` is a two-dimensional array containing 20 integers. You can use VAX C operators to form expressions with specific elements of an array, as follows:

```
++table_one[0][0];          /* Increment first element          */
```

In VAX C, arrays are stored in row-major order. The element `table_one[0][0]` immediately precedes `table_one[0][1]`, which in turn immediately precedes `table_one[0][2]`.

When you declare an array, either single- or multidimensional, the integer constant is optional in the first pair of brackets. Omitting the constant expression is useful in the following cases:

- If the array is external, its storage is allocated by a remote definition. Therefore, the constant expression can be omitted for convenience when the array name is declared, as in the following example:

```
extern int array1[];
first_function()
{
    .
    .
    .
}

/* In a separate compilation unit: */
int array1[10];
second_function()
{
    .
    .
    .
}
```

For more information about external data declarations, see Section 9.5.

- If the declaration of the array includes initializers, the size of the array can be omitted. Consider the following example:

```
char array_one[] = "Shemps"
char array_two[] = { 'S', 'h', 'e', 'm', 'p', 's', '\0' };
```

The two definitions initialize variables with identical elements. These arrays have seven elements: six characters and the null character (`'\0'`), which terminates all character strings. VAX C determines the size of the array from the number of characters in the initializing character-string constant or initialization list.

- If the array is used as a function parameter, it is defined in the calling function. The declaration of the parameter in the called function can omit the constant expression. The address of the beginning of the array is passed and subscripted references in the called function can modify elements of the array.

The following example shows how an array is used in this manner:

```
main()
{
    /* Initialize array */
    static char arg_str[] = "Thomas";
    int sum;
    sum = adder(arg_str); /* Pass address of array */
    .
    .
}

/* Function adds ASCII values of letters in array */
adder(param_string)
char param_string[];
{
    int i, sum=0; /* Incrementer and sum */
    /* Loop until NUL char */
    for (i=0; param_string[i] != '\0'; i++)
        sum += param_string[i];
    return sum;
}
```

When the function `adder` is called, parameter `param_string` receives the address of the first character of argument `arg_str`, which can then be manipulated in `adder`. The declaration of `param_string` serves only to give the type of the parameter, not to reserve storage for it.

8.7.1 Initializing Arrays

When initializing array elements, separate the values with a comma and delimit the comma-list with braces (`{}`). The rules for specifying a comma-list are as follows:

- If the initializer for an array begins with a left brace (`{}`), then the following comma-list provides initial values for the array elements. The list of initializers can end with a comma, which is ignored. The number of initializers cannot be greater than the number of elements.
- If the initializer does not begin with a left brace, then only enough elements are taken from the initializer list to supply values to the array's elements. In this case, there can be more initializers than there are elements, and any remaining values in the list are left to initialize the next aggregate.

Initialize a single-dimension array as follows:

```
int ex_array[5] = { 1, 22, 333, 4444, 55555 };
```

Initialize a multidimensional array as follows:

```
int ex_array[2][5] =
{
    { 1, 22, 333, 4444, 55555 },
    { 5, 4, 3, 2, 1 }
};
```

The element `ex_array[0][0]` has a value of 1, `ex_array[0][1]` has a value of 22, . . . , `ex_array[1][0]` has a value of 5, `ex_array[1][1]` has a value of 4, . . . , and so forth.

Another method of initializing the same array is as follows:

```
int ex_array[2][5] = { 1, 22, 333, 4444, 55555, 5, 4, 3, 2, 1 };
```

VAX C initializes the elements in row-major order. The leftmost brace determines the row number of a multidimensional array. Elements in row 0 are initialized before elements in row 1.

You may omit elements in an initialization, as follows:

```
int ex_array[2][5] =
{
    { 1, 22, 333, 4444 }
};
```

The element `ex_array[0][0]` has the value 1, `ex_array[0][1]` has the value 22, `ex_array[0][2]` has the value 333, and `ex_array[0][3]` has the value 4444. The last element in row 0, since `ex_array` was declared to have a storage class of **static**, is initialized with zero. All the elements in the second row that were not specified in the initialization are initialized with zero. For more information about the **static** storage class, see Section 9.4.

NOTE

You cannot initialize array elements without initializing all preceding elements. The following initialization is not valid:

```
example[3] = { 1 , , 3 };
```

In the previous example, you have to initialize the first and second element before initializing the third.

8.8 Character-String Variables (`char *` and `char []`)

VAX C treats character strings as arrays; they are treated as the address in memory of the first character in the string. There are several ways to declare character-string variables. You can declare a character string by designating a pointer to the first character of that string, as in the following example:

```
char *ex_string = "thomasina";
```

This expression copies an address, not a string, to variable `ex_string`. The object to which `ex_string` points, a character-string constant, ends with the NUL character (`'\0'`).

You can declare character-string variables as you would declare an array. For example:

```
char string_one[] = "thomasina";  
char string_2[10] = "thomasina";
```

To copy one string to another, you must use the **strcpy** or the **strncpy** VAX C Run-Time Library (RTL) functions, as follows:

```

main()
{
    char ex_string[26];
                                /* Copy string into array */
    strcpy(ex_string, "Character-string constant");
    printf("%s\n", ex_string);
    .
    .
}

```

For more information about the string copying VAX C RTL functions, see the *VAX C Run-Time Library Reference Manual*.

8.8.1 Character-String Constants

A *character-string constant* is a series of characters enclosed in quotation marks (" "). Consider the following example:

```
"This is a string constant *** "
```

This data type of the construct is array of **char**. The string is initialized with the given characters. The compiler terminates the string with a NUL character ('\0'). There is no formal limit to the length of a string constant. The actual limit to a string constant's length in VAX C is 65,535 characters. All strings, even when written identically, are distinct objects.

The apostrophe (') and quotation mark (") are significantly different punctuation marks in VAX C. See Section 8.3.2 for more information.

The following rules apply to the characters used in character-string constants:

- All characters, including the escape sequences, can be used in strings.
- A quotation mark within a string must be preceded by a backslash (\).
- A backslash followed immediately by a newline is ignored, allowing long strings to be continued in the first column of the next line.

8.9 Structures and Unions (struct and union)

Structures and unions are aggregates whose members can be of different types. Structures and unions are declared using the keywords **struct** and **union**, an optional tag name, and a list of member declarations delimited by braces ({ }). A member of a structure or a union is a declared segment of the data structure. The syntax for declaring a member is the same as that of any variable declaration. The structure or union tag is a name that

can be used when declaring structure or union variables of the same type elsewhere in the program.

The offset of members of the structure corresponds to the order of the member in the structure declaration. By default, the first member of a structure begins in memory at the base of the data structure, which is referred to as offset 0. Each successive nonbit-field member begins at the next byte boundary; there is no implicit type alignment. (For information about the different alignment of bit fields, see Section 8.9.5.) This alignment of structure members is a VAX C convention and is also followed by all other VAX languages. Other C implementations may align members differently. You can initialize structures at the time of declaration. (See Section 10.7.4 for information on altering default member alignment with **#pragma member_alignment**.)

Unions are declared in the same way as structures, but all members in a union begin at offset 0. Unlike structures, unions cannot be initialized. The size of the union in memory is as large as its largest member. When the single storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered.

Structures and unions share the following characteristics:

- Their members can be variables of any type, including other structures and unions or arrays. A member can also consist of a specified number of bits, called a field.
- The only valid operators used with structures and unions are the assignment (=) and **sizeof** operators. In particular, structures and unions may not appear as operands of the equality (==), inequality (!=), or cast operator.
- They can be assigned to other structures and unions with the assignment operator (=). The two structures or unions in the assignment must have the same length.
- They can be passed to and returned by functions. The argument must have the same length as the function parameter. A structure or union is passed by value, just like a scalar variable; that is, the entire structure or union is copied into the corresponding parameter.

NOTE

When you pass structures as arguments, they may or may not terminate on a longword boundary. If they do not, VAX C aligns the following argument on the next longword boundary. For more information about passing arguments between

programs written in different VMS programming languages, see Section 13.2.

The following sections discuss structures and unions in more detail.

8.9.1 Declaring a Structure or Union

Structures and unions are declared with the **struct** or **union** keywords. You can follow the **struct** or **union** keywords by a tag, which gives a name to the structure or union type in much the same way that an **enum** tag gives a name to the enumerated type. You can then use the tag with the **struct** or **union** keywords to declare variables of that type without specifying individual member declarations again.

Two structure (or two union) types cannot have the same tag, but the tags can be the same as the identifiers used for variables and function names. Tags can also have the same spellings as member names. The compiler distinguishes them by context. The scope of a tag is the same as the scope of the declaration in which it appears.

The tag is followed by braces ({ }) that enclose a list of member declarations. Each declaration in the list gives the data type and name of one or more members. The names of structure or union members can be the same as other variables, function names, or members in other structures or unions. The compiler distinguishes them by context. In addition, the scope of the member name is the same as the scope of the declaration in which it appears.

The list of member declarations can be followed by declarators, which name and reserve storage for (define) structure or union objects.

Structure or union declarations can take one of five forms, as follows:

1. If a declaration includes only a tag and a list of member declarations, then the list of member declarations defines the tag to be a data type by which other objects can be declared. For example:

```
struct person
{
    char first[20];
    char middle[3];
    char last[30];
};
```

2. When a declaration includes a tag, a list of member declarations, and a list of identifiers, the identifiers become objects of the structure type and the tag is considered to be a shorthand notation, or mnemonic, for the structure type. Consider the following example:

```
struct person
{
    char first[20];
    char middle[3];
    char last[30];
} george, mary ;
```

3. If the tag is omitted, the structure or union definition applies only to the variable identifiers that follow in the declaration. Consider the following example:

```
struct
{
    char first[20];
    char middle[3];
    char last[30];
} george, mary;
```

4. The fourth form uses the tag to see a structure or union type defined in another declaration. The definition is then applied to the variable identifiers that follow the tag name in the declaration.

```
struct person george,mary;
```

5. The fifth form uses only the **struct** or **union** keyword and the tag to override other identical tags in scope, and to reserve the tag for a later definition within a new scope. A definition within a new scope overrides any previous tag definition appearing in an outer scope. This use of declaring tags is called *vacuous structure tag declaration*. The declaration does not require the size of the structure as determined by the structure member list. Using such declarations, you can eliminate ambiguity when forward referencing tag identifiers. The following example shows such a case:

```
struct ambiguous { . . . };

{
    struct ambiguous; /* Vacuous structure tag declaration. */
                    /* Ignore previous tag currently in scope. */

    struct inner
    {
        struct ambiguous *pointer; /* Declare a structure pointer by */
        . /* forward referencing. */
        :
        .
    };
};
```

```

    struct ambiguous    /* Previous vacuous declaration refers to */
    { . . . };         /* this structure definition, not to the */
}                       /* first one declared.                */

```

In this example, the pointer to the structure defined using tag `ambiguous` points to the second declaration of `ambiguous`, not to the first.

Structures and unions can contain other structures and unions.

For example:

```

struct person
{
    char first[20];
    char middle[3];
    char last[30];
    struct
    {
        int day;
        int month;
        int year;
    } birth_date;
} george, mary;

```

8.9.2 Referencing Members of Structures or Unions

A reference to a member of a structure must be a fully qualified or a pointer-qualified reference. For example, the fully qualified references to the members `last` and `year` from the example in the previous section are as follows:

```

strcpy(george.last, "Harrison");
george.birth_date.year = 1944;

```

A member name denotes the member's data type and its offset from the base of the structure. There are no restrictions on the reuse (as a member name) or redeclaration of a particular name except that the same name cannot be used for more than one member in the same structure.

In VAX C, and in other recent compilers, a structure or union reference must be completely qualified; that is, you must prefix a member name in a reference either with a pointer qualifier (pointer-name `->`) or with the name of the structure or union and the names of all intervening members. Consider the following structure declaration:

```

main()
{
    struct
    {
        struct { int a1,a2,a3; } mema;
        struct { int a1,a2,a3; } memb;
    } *pointer, structure;
    pointer = &structure;

    structure.mema.a1 = 1;          /* Unambiguous          */
    pointer->memb.a1 = 2;

    structure.a1 = 3;              /* Ambiguous: which "a1"? */
    pointer->a1 = 4;
}

```

Member `a1` must be uniquely qualified as being a member of structure `mema` or structure `memb`. Structure members that are themselves structures must be given variable identifiers (`mema` and `memb`) to make it possible to construct fully qualified references.

A member name is unique if it conforms to either of the following requirements:

- It is used only once.
- If it is used more than once (in different structures), every use denotes a member of the same data type and at the same offset from the base of its structure.

If you use member names that refer to different structures from those in which they were declared, a programming practice not recommended, the compiler issues diagnostic messages. The following checks apply to the use of member names for reference to structures and unions in which they are not declared:

- If a member name is unique, you can use it in a reference to a structure of which it is not a member, since the address and size of the referenced data can be determined without ambiguity. However, the compiler issues a nonfatal warning message. This usage is maintained for compatibility with other C implementations.
- If a member name is not unique (ambiguous), its use in such a reference causes a fatal error message.

8.9.3 Initializing Structures and Unions

You can initialize structures at the time of declaration. You cannot initialize unions.

In structure declarations, initializers follow the structure variables, not the members. Separate initializing values with commas; delimit them with braces ({}). See Section 8.7.1 for more information about comma-lists.

An example of the initialization of two structure variables is as follows:

```
struct
{
    int i;
    float c;
} a = { 1, 3.0e10 }, b = { 2, 1.5e5 };
```

The compiler assigns structure initializers in increasing member order. If there are fewer initializers than members for a **static**, **external**, or **globaldef** structure, the structure is padded with zeros. For more information about storage classes, see Chapter 9.

NOTE

There is no way to specify iterations of an initializer or to initialize a member in the middle of a structure without also initializing the previous members.

Example 8–1 shows these initialization rules applied to an array of structures.

Example 8-1: Rules for Initialization of Structures

```
main()
{
    int l, m;
    static struct
    {
        char ch;
        int i;
        float c;
    } ar[2][3] =
    ① {
    ②     {
    ③         { 'a', 1, 3e10 },
            { 'b', 2, 4e10 },
            { 'c', 3, 5e10 },
        }
    };

    printf("row/col\t ch\t i\t c\n");
    printf("-----\n");
    for (l = 0; l < 2; l++)
        for (m = 0; m < 3; m++)
        {
            printf("[%d][%d]:", l, m);
            printf("\t %c \t %d \t %e \n",
                ar[l][m].ch, ar[l][m].i, ar[l][m].c);
        }
}
```

Key to Example 8-1:

- ① You must delimit the initialization of each of the array rows with braces.
- ② You must delimit a structure initialization with braces.
- ③ You must delimit an array initialization with braces.

This program writes the following output to stdout:

```
row/col  ch      i      c
-----
[0][0]:  a      1      3.000000e+10
[0][1]:  b      2      4.000000e+10
[0][2]:  c      3      5.000000e+10
[1][0]:  0      0      0.000000e+00
[1][1]:  0      0      0.000000e+00
[1][2]:  0      0      0.000000e+00
```

8.9.4 Variant Structures and Unions

Variant structure and union declarations allow you to reference members of nested aggregates without having to reference intermediate structure or union identifiers. When you nest a variant structure or union declaration within another structure or union declaration, the enclosed variant aggregate ceases to exist as a separate aggregate, and VAX C propagates its members to the enclosing aggregate.

You declare variant structures and unions using the keywords **variant_struct** and **variant_union**. The format of these declarations is the same as that of regular structures or unions with the following exceptions:

- Variant aggregates must be nested within other valid structure or union declarations.
- You cannot use a tag in a variant aggregate declaration.
- You must provide a variable identifier in the variant aggregate declaration.

Note that variant aggregates follow the same initialization rules as structures and unions. You can initialize both variant structures and variant unions.

To show the use of variant aggregates, consider the following code example that does not use variant aggregates:

```
/* The numbers to the right of the code represent the byte offset *
 * from the enclosing structure or union declaration.                */
struct TAG_1
{
    int    a;                /* 0-byte  enclosing_struct offset */
    char  *b;                /* 4-byte  enclosing_struct offset */
    union TAG_2              /* 8-byte  enclosing_struct offset */
    {
        int    c;            /* 0-byte  nested_union offset */
        struct TAG_3        /* 0-byte  nested_union offset */
        {
            int    d;        /* 0-byte  nested_struct offset */
            int    e;        /* 4-byte  nested_struct offset */
        } nested_struct;
    } nested_union;
} enclosing_struct;
```

If you want to access nested member *d*, then you need to specify all the intermediate aggregate identifiers, as follows:

```
enclosing_struct.nested_union.nested_struct.d
```

If you try to access member `d` without specifying the intermediate identifiers, then you would access the incorrect offset from the incorrect structure. Consider the following example:

```
enclosing_struct.d
```

VAX C uses the address of the original structure (`enclosing_struct`), and adds to it the assigned offset value for member `d` (0 bytes), even though VAX C calculated the offset value for `d` according to the nested structure (`nested_struct`). Consequently, VAX C accesses member `a` (0-byte offset from `enclosing_struct`) instead of member `d`.

The following code example shows the same code using variant aggregates:

```
/* The numbers to the right of the code present the byte offset *
 * from enclosing_struct. */
struct TAG_1
{
    int a;          /* 0-byte enclosing_struct offset */
    char *b;       /* 4-byte enclosing_struct offset */
    variant_union
    {
        int c;     /* 8-byte enclosing_struct offset */
        variant_struct
        {
            int d; /* 8-byte enclosing_struct offset */
            int e; /* 12-byte enclosing_struct offset */
        } nested_struct;
    } nested_union;
} enclosing_struct;
```

The members of variant aggregates `nested_union` and `nested_struct` are propagated to the immediately enclosing aggregate (`enclosing_struct`). The variant aggregates cease to exist as individual aggregates.

Since variant aggregates `nested_union` and `nested_struct` do not exist as individual aggregates, you cannot use tags in their declarations and you cannot use their identifiers (`nested_union`, `nested_struct`) in any reference to their members. However, you are free to use the identifiers in other declarations and definitions within your program.

If you need to access member `d`, use the following notation:

```
enclosing_struct.d
```

Using the following notation causes unpredictable results:

```
enclosing_struct.nested_union.nested_struct.d
```


If you use regular structure or union declarations within a variant aggregate declaration, VAX C propagates the structure or union to the enclosing aggregate, but the members remain a part of the nested aggregate. For instance, if the nested structure in the last example was of type **struct**, the following offsets would be in effect:

```
struct TAG_1
{
    int a;          /* 0-byte  enclosing_struct offset */
    char *b;       /* 4-byte  enclosing_struct offset */
    variant_union
    {
        int c;     /* 8-byte  enclosing_struct offset */
        struct TAG_2 /* 8-byte  enclosing_struct offset */
        {
            int d; /* 0-byte  nested_struct offset */
            int e; /* 4-byte  nested_struct offset */
        } nested_struct;
    } nested_union;
} enclosing_struct;
```

NOTE

Variant structures and unions are VAX C extensions and are not portable.

8.9.5 Bit Fields

A structure member may consist of a specified number of bits, called a field, which may be named or unnamed. A colon is used to separate the member's name (if any) from a constant-expression that gives the field width in bits. No field may be longer than 32 bits (1 longword) in VAX C.

If no field name precedes the field-width expression, it indicates an unnamed field of the specified width. Since, by default, nonfield structure members are aligned on byte boundaries, this form can create unnamed gaps in the structure's storage. As a special case, an unnamed field of width 0 causes the next member (generally another field) to be aligned on a byte boundary. (See Section 10.7.4 for information on altering default member alignment with **#pragma member_alignment**.)

Bit fields must be of data types **unsigned** or **int**. The use of other data types is an error. Signed bit fields of the type **int** are recognized by VAX C. There are no restrictions on the use of fields except as follows:

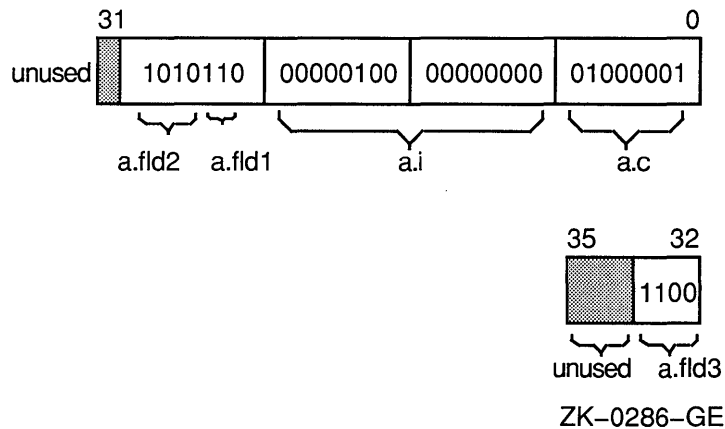
- You cannot declare arrays of fields.
- The address-of operator (&) cannot be applied to fields, and consequently there cannot be pointers to fields.

Constructs of all data types except fields are aligned on the next byte boundary. Sequences of bit fields are packed as tightly as possible. In VAX C, fields are assigned from right to left.

For example, Figure 8-1 shows the alignments resulting from the following code:

```
static struct
{
    char c;
    short int i;
    unsigned fld1 : 3;
    unsigned fld2 : 4;
    unsigned      : 0;
    unsigned fld3 : 4;
} a = { 'A', 1024, 06, 012, 014 } ;
```

Figure 8-1: Alignment of Structure Members



In Figure 8–1, member `a.i` is aligned on the second byte (at bit 8), because scalar, nonfield members are aligned on byte boundaries. Notice that fields `a.fld1` and `a.fld2` are packed as tightly as possible in the high-order byte of the first longword. The unnamed, zero-length field preceding member `a.fld3` causes that field to be aligned on the next byte boundary (bit 32, the second longword).

8.10 The `void` Keyword

The **`void`** keyword is a special data-type specifier that you use in function definitions and declarations for the following purposes:

- To specify a function that does not return a value.
- To specify a function prototype with no arguments.
- To specify a pointer variable to an unspecified or unknown data type. (See Section 8.5.1 for more information.)

For instance, the following example shows how to use **`void`** to specify a function that does not return a value:

```
void message( )
{
    printf("Stop making sense!");
    return;
}
```

The following example shows how to use **`void`** to specify a function prototype definition that takes no arguments:

```
char function_name( void )
{ return 'a'; }
```

For more information about the **`void`** data type and function prototypes, see Sections 5.2 and 5.3.

8.11 The `typedef` Keyword

The **`typedef`** keyword is used to define an abbreviated name, or synonym, for a lengthy type definition. In such a declaration, the identifiers name types instead of variables. For example:

```
typedef char CH, *CP, STRING[10], CF();
```

In the scope of this declaration, CH is a synonym for character, CP is a synonym for pointer to character, STRING is a synonym for a 10-element array of characters, and CF is a synonym for a function returning a character. Each of the type definitions can be used in that scope to declare variables, as follows:

```
CF      c;          /* "c": Function returning a character */
STRING s;          /* "s": 10-character string */
```

8.12 Interpreting Declarations

The VAX C programming language syntax for declaring objects is unlike the declaration syntax of other languages. Since the exact meaning of a complicated VAX C declaration is not immediately apparent, even to an experienced C programmer, this section gives you guidelines for interpreting and constructing VAX C declarations.

One way to interpret a complex declaration is to compile your program using the `/LIST/SHOW=SYMBOLS` qualifiers. The generated symbol map contains an English description of the type of each variable.

VAX C uses the same set of operators and symbols for declarators as for identifiers in an expression. For example, the following example declares integer `x` and pointer `px`:

```
int  x;
int  *px;
```

Declarator `*px` has the same form as that used to yield an integer in an expression. For example:

```
x = *px;
```

In the case of simple declarators, this symmetry makes it easy to determine the type of an expression or the meaning of a declarator. Expression `*px` results in the integer object to which `px` points.

Complicated declarators can be difficult to interpret without some additional guidelines. The important one to remember is that the symbols used in declarators are VAX C operators, subject to the usual rules of precedence and grouping (associative nature). In order of precedence, the operators used in declarators are as follows:

1. The primary-expression operators `(())` for “function returning . . . ” and `[]` for “array of . . . ”, where the ellipsis indicates the type specified in the declaration.

These operators group from left to right.

2. The unary asterisk (*), for indirection or “pointer to . . .”, which groups from right to left.

Consider the following example:

```
int *x[];
```

Even this brief declaration may be confusing. Does it declare an array of pointers to integers, or a pointer to an array of integers? Since the brackets are of higher precedence, it follows that:

1. *x[] is an integer
2. x[] is a pointer to an integer
3. x is an array of pointers to integers

Most complicated declarators and expressions can be interpreted quickly by such a sequential breakdown. Note that the asterisk was removed before the brackets because it is of lower precedence.

Also note that this interpretation process has the desirable property of enumerating all the possible usage constructs involving a declarator and giving the semantic interpretation.

When constructing or interpreting declarations or expressions, use the following scheme¹ for translating operators to English and vice versa:

- “*” == “pointer to”
- “()” == “function returning”
- “[]” == “array of”

Consider the following example:

```
char *x() [];
```

The breakdown is as follows:

1. *x()[] is **char**
2. x()[] is (pointer to) **char**
3. x() is (array of) (pointer to) **char**
4. x is (function returning) (array of) (pointer to) **char**

In step 3, the brackets operator is removed first because primary-expression operators are of equal precedence and group from left to right. That is, “()[]” means “function returning array of”, not “array of function returning . . .”.

¹ Bruce Anderson, “Type Syntax in the Language C: An Object Lesson in Syntactic Innovation,” *SIGPLAN Notices* 15, No. 2 (March 1980).

As a general rule, when breaking down a declaration this way, remove the operators with the lowest precedence first. Then, if operators are of equal precedence and group from left to right, remove the rightmost operator first; if they group from right to left, remove the leftmost operator first.

The declaration shown is semantically invalid; VAX C allows functions returning addresses of arrays, but not functions returning arrays. Perhaps the intention of the programmer was a function returning the address of an array of pointers to characters. The declaration can be made valid by starting at the bottom of a breakdown and working back to a valid declaration as follows:

1. `x` is (function returning) (pointer to) (array of) (pointer to) **char**
2. `x()` is (pointer to) (array of) (pointer to) **char**
3. `*x()` is (array of) (pointer to) **char**
4. `(*x())[]` is (pointer to) **char**
5. `*(*x())[]` is **char**
6. `char *(*x())[];`

In the final declaration, the first asterisk (since it groups right to left) applies to **char**.

Parentheses, in addition to the function parameter-list operator `(())`, are used in declarations to change the binding of operators. For example, the outer parentheses introduced in step 4 of the previous example prevent the brackets from binding to the inner set of parentheses.

Consider the following example:

```
char (* (*x()) []) ();
```

The breakdown is as follows:

1. `(* (*x()) []) ()` is **char**
2. `* (*x()) []` is (function returning) **char**
3. `(*x()) []` is (pointer to) (function returning) **char**
4. `*x()` is (array of) (pointer to) (function returning) **char**
5. `x()` is (pointer to) (array of) (pointer to) (function returning) **char**
6. The identifier `x` is a function returning a pointer to an array of pointers to functions returning characters

Spaces were used in the example to separate the declarator into its component parts. Since spaces, tabs, and newlines are ignored by the parser, use them in declarations for clarity.

Storage Classes and Allocation

The VAX C language defines a number of storage-class keywords that specify the scope of an identifier, the location of storage, and the lifetime of the storage allocation. Storage-class modifiers are keywords that you can use with the storage-class and data-type keywords that restrict access to variables. The order of the storage-class keyword, the storage-class modifier, the data-type modifier, and the data-type keyword within the variable declaration does not matter. Each declaration, by virtue of its position in the program source code, has a default storage class, but you may override the default by specifying a storage-class specifier or a storage-class modifier.

This chapter discusses the following topics:

- Scope of an identifier (Section 9.1)
- Storage allocation (Section 9.2)
- Internal storage classes (Section 9.3)
- Static storage class (Section 9.4)
- External storage class (Section 9.5)
- Global storage classes (Section 9.6)
- Data-type modifiers (Section 9.7)
- Storage-class modifiers (Section 9.8)

9.1 The Scope of an Identifier

The scope of an identifier is the portion of the program in which the identifier has meaning. An identifier has meaning if it is recognized by the compiler, or by the VMS Linker. The following sections explain the rules to follow so that your program identifiers will have meaning, to both the compiler and the linker, in all desired portions of your program.

All tags are subject to the same scope rules as other identifiers. A member of a structure or union may have the same name as a member of another structure or union; the scope of the member names can exist concurrently. However, when referencing one of the members in a section of the program where the scopes of both members are concurrent, take care to specify to which structure or union the member belongs. For more information, see Chapter 8.

9.1.1 The Compilation and Linking Process

To understand scope, you must understand the VMS definitions of function, compilation unit, object file, object module, and program.

When you write VAX C source programs, you can use several methods to compile a program. You can compile a single source file, or a group of source files, into a single object file. The group of source file(s) compiled to create a single object file is called the *compilation unit*. When documentation to other implementations refers to the source file, the VMS equivalent is the compilation unit, not necessarily a single source file. The single, resultant object file has a file extension of .OBJ, by default.

The linker accepts the object file as input and then resolves all external references, such as references to VAX C Run-Time Library (RTL) functions. Internally, segments of object code, such as the object file and the VAX C RTL object code, are known to the linker as object modules. The object module has the same name (without an extension) as the object file, by default. For information on how to override the default module name, see Section 10.6.

Another way to compile programs is to compile several compilation units into separate object files. The linker can take more than one object file as input. In this case, the linker resolves references between modules specified on the command line and to external references in your program. For more information about compiling and compilation units, see Section 1.3. For more information about linking, see Section 1.4.

9.1.2 Position of the Declaration

To determine the scope of a function or variable identifier, you must consider the position of a declaration within the program. A declaration often determines the size of a storage allocation, but a definition initiates the allocation of storage. Since declarations often are definitions, this section refers to definitions and declarations as declarations. You may wish to review Chapter 8 before reading the rest of this section.

The location of a declaration establishes the scope of an identifier. If a declaration is located inside a block that is delimited by braces ({}), the compiler recognizes the identifier from the point of the declaration to the end of the block. If a declaration is located outside of all functions, the compiler recognizes the identifier from the point of the declaration to the end of the compilation unit.

You can specify a storage-class specifier or modifier within an identifier's declaration. A storage-class specifier indicates a storage class, but a modifier modifies access to that storage. The order of the storage-class specifier, storage-class modifier, and the data-type keyword within the declaration does not matter. Consider the following example:

```
auto int x;    /* And, equivalently . . . */
int auto x;
```

You can declare identifiers that are one of the internal storage classes; the compiler recognizes these identifiers from the point of the declaration to the end of the enclosing block or function body. You can declare identifiers that are static; if the declaration is outside of all function bodies, the compiler recognizes these identifiers from the point of the declaration to the end of the compilation unit.

You can also declare identifiers that are of the external storage classes or of the global storage classes. If the declaration is outside of all function bodies, the compiler recognizes these identifiers from the point of the declaration to the end of the compilation unit. The external and global storage classes differ from the static storage class in that the linker can possibly recognize a global or external variable from the point of the declaration to the end of the program. The external and global storage classes establish a scope that can possibly span object modules. The external and global storage classes are also different from each other. See Section 9.6.1.1 for a comparison between the global and external storage classes.

Table 9–1 presents the list of storage classes followed by the storage-class specifiers used to establish scope.

Table 9–1: VAX C Storage Classes and Storage-Class Specifiers

Storage Class	Specifiers
Internal	auto, register , absence of specifier inside a block or function ¹
Static	static
External	extern , absence of specifier outside of all functions
Global	globaldef, globalref, globalvalue

¹Functions declared without a storage-class specifier are of the external storage class, by default.

For more information about the internal storage classes, see Section 9.3. For more information about the static storage class, see Section 9.4. For more information about the external storage classes, see Section 9.5. For more information about the global storage classes, see Section 9.6.

You can use the data-type modifiers (**const** and **volatile**) or the VAX C specific storage-class modifiers (**readonly** and **noshare**) to restrict access to data or to specify storage requirements. See Section 9.7 for more information about the data-type modifiers. See Section 9.8 for more information about the storage-class modifiers.

9.1.3 Lexical Scope and Link-Time Scope

In using the storage-class specifiers and modifiers, as well as positioning the definitions and declarations of your identifiers, keep the following two goals in mind:

- Compile the program so that the compiler recognizes all identifiers in the compilation unit, thus avoiding error messages.
- Link the program so that the VMS Linker resolves all references to external data definitions, thus avoiding error messages.

You must make a distinction between the following types of scope:

Lexical scope	The region of a compilation unit within which an identifier is known to the compiler. When this guide uses the term scope, lexical scope is implied.
---------------	--

Link-time scope The regions of an entire program within which an external or global identifier is known to the linker. Only the identifiers in the external and global storage classes have a significant link-time scope.

Table 9–2 lists the VAX C storage-class specifiers and shows both the link-time scope and lexical scope implied by each specifier when used inside and outside of functions.

Table 9–2: Scope and the Storage-Class Specifiers

Storage Class	Inside a Function		Outside a Function	
	Lexical Scope	Link-time Scope	Lexical Scope	Link-time Scope
[auto]	function	N/A	illegal	illegal
register	function	N/A	illegal	illegal
static	function	function	CU ¹	module
[extern]	function	program	CU ¹	program
globaldef	function	program	CU ¹	program
globalref	function	program	CU ¹	program
globalvalue	function	program	CU ¹	program
(null)	function	N/A	CU ¹	program

¹Compilation Unit

Identifier (null) signifies the absence of a storage-class specifier from the declaration. The compiler treats a (null) inside a function or block as an identifier declared with the **auto** keyword. The compiler treats a (null) outside all functions as an external definition, the identifier being of the external storage class.

In Table 9–2, the notation **[auto]** specifies identifiers of the automatic storage class. If you do not include a storage-class specifier on a definition inside of a function definition, the object is **auto** by default. This notation is used throughout this manual to represent the automatic storage class, regardless of the presence of the **auto** specifier in the definition.

Also in Table 9–2, the notation **[extern]** signifies identifiers of the external storage class. A single definition exists without the use of a storage-class specifier; other declarations, which use the **extern** specifier, may exist that reference that definition. This notation is used throughout this manual to represent the external storage class, regardless of the presence of the

extern specifier in the declaration or definition. See Section 9.5 for more information about the external storage class.

9.1.4 Program Example

Determining the scope of **static**, external, and global symbols can be very difficult. Consider Example 9–1.

Example 9–1: Scope and Externally Defined Variables

Compilation Unit 1	Compilation Unit 2
-----	-----
<code>globaldef int GLOBAL_1;</code>	
<code>int EXT_2;</code>	<code>int EXT_1;</code>
<code>static int STAT;</code>	
<code>f1()</code>	<code>f3()</code>
<code>{</code>	<code>{</code>
<code>globaldef int GLOBAL_2;</code>	<code>extern int EXT_2;</code>
<code>·</code>	<code>·</code>
<code>·</code>	<code>·</code>
<code>·</code>	<code>·</code>
<code>}</code>	<code>}</code>
<code>extern int EXT_1;</code>	<code>globalref int GLOBAL_2;</code>
<code>f2()</code>	<code>f4()</code>
<code>{</code>	<code>{</code>
<code>·</code>	<code>globalref int GLOBAL_1;</code>
<code>·</code>	<code>·</code>
<code>·</code>	<code>·</code>
<code>}</code>	<code>}</code>
	<code>f5()</code>
	<code>{</code>
	<code>static int STAT;</code>
	<code>·</code>
	<code>·</code>
	<code>·</code>
	<code>}</code>

Table 9–3 specifies the variable identifiers in Example 9–1, and in which functions they can be accessed without compile-time errors.

Table 9–3: The Variables in Example 9–1 and Their Storage Classes

Identifier	Scope
GLOBAL_1	<p>This variable is defined outside of all functions in Compilation Unit 1, so you can access GLOBAL_1 in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit).</p> <p>In Compilation Unit 2, the declaration of this variable is located inside of function f4; the scope of the variable, in this compilation unit, only extends from the declaration of GLOBAL_1 to the end of function f4.</p>
GLOBAL_2	<p>This variable is defined inside the function f1. In Compilation Unit 1, the scope of GLOBAL_2 only extends from the declaration of GLOBAL_2 to the end of function f1.</p> <p>In Compilation Unit 2, the declaration of this variable is outside of all functions but is located after the function f3; you can access the variable in the functions f4 and f5 (from the point of the declaration to the end of the compilation unit).</p>
EXT_1	<p>This variable is declared outside of all functions. This declaration is a reference to the definition of the same variable in the other compilation unit. In Compilation Unit 1, you can access EXT_1 in the function f2 (from the point of the declaration to the end of the compilation unit).</p> <p>In Compilation Unit 2, the definition of this variable is outside of all functions; you can access EXT_1 in the functions f3, f4, and f5 (from the point of the declaration to the end of the compilation unit).</p>
EXT_2	<p>This variable is defined outside of all functions. In Compilation Unit 1, you can access EXT_2 in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit).</p> <p>In Compilation Unit 2, the declaration of this variable is located inside of the function f3; you can access EXT_2 from the location of this declaration to the end of function f3.</p>

(continued on next page)

Table 9–3 (Cont.): The Variables in Example 9–1 and Their Storage Classes

Identifier	Scope
STAT	<p>There are two variables with the same name but with different permanent storage locations. In essence, these are two different variables.</p> <p>In Compilation Unit 1, the variable is defined outside of all functions. You can access STAT, in Compilation Unit 1, in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit).</p> <p>In Compilation Unit 2, the separate variable is defined inside of the function f5; you can access STAT from this declaration to the end of the function f5.</p>

Another way to look at the determination of scope is to see the placement of the declaration as a matter of privacy. In Compilation Unit 2 in Table 9–3, identifier EXT_2 is made private to function f3 by placing the declaration inside the function body. If you want to keep a variable private to Compilation Unit 1, use a declaration using the storage-class specifier **static**; there is no way to access a variable declared with **static** in another compilation unit. Using the storage-class specifiers **auto** and **register** ensures privacy to the function, since these specifiers cannot be used outside a function body and storage is deallocated at the end of execution of the containing function body. Similarly, there is no way to access a variable declared with **auto** or **register** in another function or compilation unit.

9.2 Storage Allocation

When you define a variable, the storage class determines not only its scope but also its location and lifetime. The lifetime of a variable is the length of time for which storage is allocated. Storage for a variable can be allocated in the following locations:

- On the run-time stack
- In a machine register
- In a program section (psect)

Variables that are placed on the stack or in a register are temporary. For example, the variables of storage class **auto** and **register** are temporary. Their lifetimes are limited to the execution of a single block or function. All declarations of the internal storage class are also definitions; the compiler generates code to establish storage at this point in the program.

Program sections, or psects, are used for permanent variables; the identifier's lifetimes extend through the course of the entire program. A psect represents an area of virtual memory that has a name, a size, and a series of attributes that describe the intended or permitted usage of that portion of memory. For example, the compiler places variables of the static, external, and global storage classes in psects; you have some control as to which psects contain which identifiers. All declarations of the static storage class are also definitions; the compiler creates the psect at that point in the program. In VAX C, the first declaration of the external storage class is also a definition; the linker initializes the psect at that point in the program.

Table 9-4 shows the location and lifetime of a variable when you use each of the storage-class keywords.

Table 9-4: Location, Lifetime, and the Storage-Class Keywords

Storage Class	Location	Lifetime
(Internal null)	Stack or register	Temporary
[auto]	Stack or register	Temporary
register	Stack or register	Temporary
static	Psect	Permanent
[extern]	Psect	Permanent
globaldef	Psect	Permanent
globalref	Psect	Permanent
globalvalue	No storage allocated	Permanent

For detailed information about psects, see Chapter 14. For information about the functional differences between the **extern** psects and the **globaldef** and **globalref** psects, see Section 9.6.1.1.

9.3 Internal Storage Classes

You can assign an internal storage class to identifiers using the **auto** and **register** storage-class specifiers. The following sections describe these specifiers.

9.3.1 The auto Specifier

Use the **auto** storage-class specifier to define a variable whose storage is automatically allocated upon entry into a function or block, and is automatically deallocated upon exit from a function or block. The code generated by the compiler contains instructions to allocate and deallocate the storage by using machine registers and the run-time stack. Since new storage allocation occurs upon entering a block or function, you can have more than one **auto** variable with the same name as long as you declare them in separate blocks or functions. You cannot use **auto** outside of a function.

If you explicitly initialize an **auto** variable, the program code initializes the variable to that value each time the declaring block or function is activated normally. This initialization cannot occur if control passes into a block by some other means, such as a **goto** statement or if the block is the body of a **switch** statement. For more information about the **switch** and **goto** statements, see Chapter 6.

Within a function, **auto** is the default storage class. That is, any variable (other than a function name) declared within a function without a storage-class specifier is given the **auto** storage class. Functions are of the external storage class by default.

NOTE

The compiler assigns **auto** variables to machine registers, if `/OPTIMIZE` is in effect and if possible. Otherwise, they are placed on the run-time stack.

Example 9-2 shows how to reinitialize two **auto** variables with the same name.

Example 9–2: Reinitializing Two auto Variables

```
/* This example prints the values of two distinct auto   *
 * variables that have the same identifier.             */
main()
{
  ❶ int i, x = 2;
    printf("main: %d\n",x);
    for (i = 0; i < 1; i++)
      ❷ {
        int x = 3;
        printf("for loop: %d\n",x);
      }
    printf("main: %d\n", x);
}
```

Key to Example 9–2:

- ❶ This definition of variable `x` extends through the entire function.
- ❷ This definition of variable `x` is limited to the **for** statement and supersedes the value of variable `x` in the surrounding function.

The output for Example 9–2 is as follows:

```
$ RUN EXAMPLE.EXE RETURN
main: 2
for loop: 3
main: 2
```

In this program, the variable `x` is defined twice within the `main` function, but the two variables do not conflict. While the **for** loop is executing, the variable `x` declared inside the block supersedes the variable `x` declared outside the block.

9.3.2 The register Specifier

Variables declared with the **register** storage class are similar to **auto** variables. You can only use the **register** internal storage class inside functions and blocks.

NOTE

The **register** storage-class specifier is the only specifier that you can use in a parameter declaration.

A **register** variable differs from a variable of storage class **auto** in the way that compiler-generated program code allocates storage. The **register** storage-class keyword suggests that the compiler flag the variable for placement in a machine register. This does not guarantee that the program code will place the variable in a register. The compiler checks the following conditions to determine whether or not a variable is flagged to be placed in a register:

- If the variable is not used, the optimizer may remove it entirely.
- If the program is compiled with the `/NOOPTIMIZATION` command qualifier, no variables are assigned to registers. The optimization phase of the compiler determines whether a variable is a valid candidate for a register.
- If the program contains too many register candidates, not all of them are assigned to registers.
- If the compiler detects any use of the variable that may make it inappropriate for assignment to a register, the variable is not flagged. For example, if the compiler detects the application of the address-of operator (`&`) to a variable that was declared with the **register** specifier, the variable is not placed in a register.

9.4 Static Storage Class

The static storage class allows you to create permanent storage for a variable using the **static** storage-class specifier in the variable declaration. If declared inside a function, its scope begins at the declaration and spans the body of the function. If declared outside of all functions, its scope is limited to the compilation unit; you cannot access a variable of the static storage class from another compilation unit.

If no initialization is present in the declaration of a variable of the static storage class, the linker initializes the variable to zero. However, unlike **auto** variables, the compiler-generated program code does not reallocate storage for a **static** variable every time control reenters a function containing the definition of a **static** variable. That is, if when exiting a function a **static** integer variable has the value of 4, the variable retains that value even if control reenters the defining function. If a **static** identifier with the same name is declared in another module, the linker knows nothing of the other variable; the other variable has a separate psect allocation.

A function can also be defined with the **static** storage-class specifier. A **static** function is not known to the linker and can be referenced only from within its defining module.

For more information about the possible combinations of specifiers and modifiers, and the effects of the storage-class modifiers on program section attributes, see Chapter 14.

9.5 External Storage Class

You can declare identifiers of the external storage class in the following manner:

- A definition not using another storage-class keyword, located outside of all function bodies, establishes an external variable whose scope extends from the point of the definition to the end of the compilation unit.
- A declaration using the **extern** keyword, usually located in another compilation unit, is a reference to the original definition. This declaration extends the link-time scope of the variable into the second object module. If this declaration is inside a function, it extends the link-time scope from the point of the declaration to the end of the function. If this declaration is outside a function, it extends the link-time scope from the point of the declaration to the end of the object module.
- You do not always *have* to use external variable declarations (with the **extern** keyword) to refer to the definition of an external variable. You can also use more than one **extern** declaration to reference the external definition.

Use the following rules when deciding whether or not to use the **extern** specifier:

- If the variable is defined before it is referenced and the definition is in the same compilation unit, you do not need to declare the variable with the **extern** specifier.
- If the variable is defined after it is referenced, you need to first declare it with the **extern** specifier.
- If the variable is defined in a separate compilation unit, you must declare it with the **extern** specifier.

Consider the following example:

```
double D = 2.37;

main()
{
    extern int A;

    printf("a:\t%d\n", A);
    printf("d:\t%g\n", D);
}

int A = 5;
```

The main function in this program references two external variables, A and D. Since the variable D is defined before it is referenced, it does not have to be declared in the main function. Since the variable A is referenced before it is defined, it must be declared with the **extern** storage-class specifier.

In many implementations of the C language, you cannot use the **extern** specifier in a declaration that does not refer to an external definition elsewhere in the program. Whenever the compiler encounters the first declaration of an identifier of the external storage class in a VAX C program, it creates and initializes a psect. Therefore, in VAX C, you can use the **extern** specifier in a declaration that does not refer to an external definition elsewhere in the program. This is not good programming practice and, if used, your programs may not be portable to other systems.

NOTE

In VAX C, you *cannot* initialize an identifier declared with the **extern** specifier.

External variables occupy storage in psects of the same name as the variable identifier. When the linker manages the psects of the external variables, the identifiers, no matter how they appear in the source code, appear in uppercase to the linker. It is good programming practice to express all external variables (and global variables as well) in uppercase letters. This practice aids the debugging of your programs.

You can specify the **noshare** modifier with external variables to create a psect with the NOSHR attribute. Similarly, you can specify the **readonly** or **const** modifier to create a NOWRT psect. For more information about the possible combinations of specifiers and modifiers, and the effects of the storage-class modifiers on program section attributes, see Chapter 14.

9.6 Global Storage Classes

You can assign the global storage class to identifiers using the **globaldef**, **globalref**, or **globalvalue** storage-class specifiers. The following sections describe these specifiers.

9.6.1 The **globaldef** and **globalref** Specifiers

Use the **globaldef** specifier in the definition of a global variable; use the **globalref** specifier in reference to a global variable defined elsewhere in the program. The **globaldef** and **globalref** specifiers are used in much the same way as with the external storage class. Use **globalref** to refer to storage allocated elsewhere by a **globaldef** declaration.

When defining a global symbol using the **globaldef** specifier, place the symbol in one of three program sections: the \$DATA psect (**globaldef** alone), the \$CODE psect (**globaldef** with **readonly** or **const**), or a user-named psect. You can create a user-named psect by specifying the psect name as a string constant in braces immediately following the **globaldef** keyword, as shown in the following definition:

```
globaldef{"psect_name"} int x = 2;
```

This definition creates a program section called `psect_name` and allocates the variable `x` in that psect. You can add any number of global variables to this psect by specifying the same psect name in other **globaldef** declarations. In addition, you can specify the **noshare** modifier to create the psect with the NOSHR attribute. Similarly, you can specify the **readonly** or **const** modifier to create the psect with the NOWRT attribute. For more information about the possible combinations of specifiers and modifiers, and the effects of the storage-class modifiers on program section attributes, see Chapter 14.

Variables declared with **globaldef** may be initialized; variables declared with **globalref** may not, since these declarations refer to variables defined, and possibly initialized, elsewhere in the program. Initialization is possible only when storage is allocated for an object. This distinction is especially important when the **readonly** or **const** modifier is used; unless the global variable is initialized when the variable is defined, its permanent value is 0.

NOTE

In the VAX MACRO programming language, it is possible to give a global variable more than one name. However, in VAX C, only one global name can be used for a particular variable. VAX C assumes

that distinct global variable names denote distinct objects; the storage associated with different names must not overlap.

Example 9-3 shows the use of global variables.

Example 9-3: Using Global Variables

```
/* This example shows how global variables are used      *
 * in VAX C programs.                                   */

❶ int ex_counter = 0;
❷ globaldef double velocity = 3.0e10;
❸ globaldef {"distance"} long miles = 100;

main()
{
    printf("    *** FIRST COMP UNIT ***\n");
    printf("counter:\t%d\n", ex_counter);
    printf("velocity:\t%g\n", velocity);
    printf("miles:\t\t%d\n\n", miles);
    fn();
    printf("    *** FIRST COMP UNIT ***\n");
    printf("counter:\t%d\n", ex_counter);

❹    printf("velocity:\t%g\n", velocity);
    printf("miles:\t\t%d\n\n", miles);
}

/* ----- *
 * The following code is contained in a separate *
 * compilation unit.                             *
 * ----- */

static ex_counter;
❺ globalref double velocity;
  globalref long miles;

fn()
{
    ++ex_counter;
    printf("    *** SECOND COMP UNIT ***\n");
    if ( miles > 50 )
        velocity = miles * 3.1 / 200 ;
    printf("counter:\t%d\n", ex_counter);
    printf("velocity:\t%g\n", velocity);
    printf("miles:\t\t%d\n", miles);
}
```

Key to Example 9-3:

- ❶ The integer variable `ex_counter` is a variable of storage class **extern** in the first compilation unit. In the second compilation unit, a variable `ex_counter` is of storage class **static**. Even though they have the same

identifier, the two `ex_counter` variables are different variables represented by two separate memory locations. The link-time scope of the second `ex_counter` is the module created from the second compilation unit. When control returns to the main function, the external variable `ex_counter` retains its original value.

- ② The variable `velocity` is a variable of storage class **globaldef** and is stored in the psect `$DATA`.
- ③ The variable `miles` is also a variable of storage class **globaldef** but is stored in the user-specified psect “distance”.
- ④ When the variable `velocity` prints after the function `fn` executes, the value will have changed. Global variables have only one storage location.
- ⑤ When you reference global variables in another module, you must declare those variables in that module. In the second module, the global variables are declared with the **globalref** keyword.

Sample output from Example 9–3 is as follows:

```
$ RUN EXAMPLE.EXE RETURN
*** FIRST COMP UNIT ***
counter:      0
velocity:    3.000000e+10
miles:       100
*** SECOND COMP UNIT ***
counter:      1
velocity:    1.55
miles:       100
*** FIRST COMP UNIT ***
counter:      0
velocity:    1.55
miles:       100
```

9.6.1.1 Comparing the Global and the External Storage Classes

The global storage-class specifiers define and declare objects that differ from external variables both in their storage allocation and in their correspondence to elements of other languages. Global variables provide a convenient and efficient way for a VAX C function to communicate with assembly language programs, with VMS system services and data structures, and with other high-level languages that support global symbol definition, such as VAX PL/I. For more information about multilanguage programming, see Chapter 13.

VAX C imposes no limit on the number of external variables in a single program.

NOTE

The global storage classes are VAX C specific and are not portable.

There are other functional differences between the external and global variables. For example:

- If you have a limited amount of storage available, you may use the **globalvalue** specifier (see Section 9.6.2) since it does not occupy storage in your program if expressed in 32 or fewer bits; the external variables create program sections.
- You can declare a global variable, using **globaldef**, inside a function or block, and by using a **globalref** specifier, access the identifier from another compilation unit. With external variables, you must define the variable outside all functions and blocks, and then access that variable in other compilation units by using **extern** declarations.
- The global variables correspond to global symbols declared in assembly language programs but external variables (**extern**) correspond with FORTRAN common blocks.
- A **globalref** declaration causes the linker to load the module containing the corresponding **globaldef** into the image. An **extern** declaration does not cause the linker to do so. An **extern** declaration causes an overlaying of a psect (see Chapter 14 for details about psects).

In programming environments other than the VMS system, C programmers may be accustomed to **extern** declarations causing the loading of a module into the program's executable image. If transportability is an issue, you can define the following symbols—at the compilation-unit level, outside of all functions—to allocate storage differently depending on the system you are using:

```
#ifdef VAXC
#define EXPORT globaldef
#define IMPORT globalref
#else
#define EXPORT
#define IMPORT extern
#endif
.
.
.
IMPORT int foo;
EXPORT int foo = 53;
```

One similarity between the external and global storage classes is in the way the linker recognizes these variables internally. No matter how the external and global identifiers appear in the source code, the linker converts these identifiers to uppercase letters. For ease in debugging programs, express all global and external variable identifiers in uppercase letters.

Another similarity between the external and global storage classes is that you can place the external variables (by default) and the global variables (optionally) in psects with a user-defined name and, to some degree, user-defined attributes. The compiler places external variables in psects of the same name as the variable identifier, viewed by the linker in uppercase letters. The compiler places **globaldef**("name") variables in psects with names specified in quotation marks, delimited by braces, and located directly after the **globaldef** specifier in a declaration. Again, the linker considers the psect name to be uppercase letters.

The compiler places a variable declared using only the **globaldef** specifier and a data-type keyword into the \$DATA psect. For more information about the possible combinations of specifiers and modifiers, and the effects of the storage-class modifiers on program section attributes, see Chapter 14.

9.6.2 The globalvalue Specifier

A global value is an integral value whose identifier is a global symbol. Global values are useful because they allow many programmers in the same environment to refer to values by identifier, without regard to the actual value associated with the identifier. The actual values can change, as dictated by general system requirements, without requiring changes in all the programs that refer to them. If you make changes to the global value, you only have to recompile the defining compilation unit (unless it is defined in an object library), not all of the compilation units in the program that refer to those definitions.

NOTE

You can use the **globalvalue** specifier only with variables of type **enum**, **int**, or with pointer variables.

A variable declared with **globalvalue** does not require storage. Instead, the linker resolves all references to the value. If an initializer appears with **globalvalue**, the name defines a global symbol for the given initial value. If no initializer appears, the **globalvalue** construct is considered a reference to some previously defined global value.

Predefined global values serve many purposes in VMS system programming, such as defining status values. It is customary in VMS system programming to avoid explicit references to such values as those returned by system services, and to use instead the global names for those values. Example 9-4 shows the use of the **globalvalue** storage-class specifier.

Example 9–4: Using the globalvalue Specifier

```
/* This program shows references to previously defined      *
 * globalvalue symbols.                                   */

globalvalue FAILURE = 0, EOF = -1;

main()
{
    char c;
    while ( (c = getchar()) != EOF) /* Get a char from stdin */
        test(c);
}

/* ----- *
 * The following code is contained in a separate compilation *
 * unit. *
 * ----- */

#include ctype /* Include proper module */
globalvalue FAILURE, EOF; /* Declare global symbols */

test(param_c)
char param_c; /* Declare parameter */
{
    /* Test to see if number is valid */
    if ( (isalnum(param_c)) != FAILURE)
        printf("%c\n", param_c);
    return;
}
```

In Example 9–4, FAILURE and EOF are defined in the first module: the values are placed into the program stream. In the second module, FAILURE and EOF are declared so that their values may be accessed. Like the external and global variables, the linker recognizes the global symbols as uppercase letters. Express these symbols in uppercase.

9.6.3 Global Enumerated Types

When you use the **globaldef** storage-class keyword with an **enum** definition, the enumerated constants in the definition are of the storage class **globalvalue**, initialized as the program requires to form a properly ordered list of the values. Enumerated type variables are of the storage class **globaldef**.

When you use **globalref** with the **enum** keyword, all enumerated variables are of the storage class **globalref**, and the enumerated constants refer to global values of the same names as shown in the following example.

The first compilation unit is as follows:

```
globaldef enum light { dim, medium=3, bright } light_val;
main()
{
    light_val = dim;
    fnlv();
}
```

The second compilation unit is as follows:

```
globalref enum light { dim, medium, bright } light_val;
fnlv()
{
    if (light_val < bright ) printf("TOO DIM\n");
}
```

In the first compilation unit, the **enum** definition establishes `light_val` as a **globaldef** of the enumerated type `light`. It also establishes the ordered list of enumerated global values `dim`, `medium`, and `bright`.

The **globalref** declaration in the second compilation unit allows the enumerated constants to be used as global values. That is, the constants can be referenced, but not initialized.

For more information about the enumerated data type, see Section 8.6.

9.7 Data-Type Modifiers

Data-type modifiers affect the allocation or access of data storage. The data-type modifiers are as follows:

- **const**
- **volatile**

The following sections describe the data-type modifiers in detail.

9.7.1 The **const** Modifier

The **const** data-type modifier restricts access to stored data. If you declare an object to be of type **const**, you cannot modify that object.

The following rules apply to the use of the **const** data-type modifier:

- You can specify **const** with any of the other data-type keywords in a declaration.

- If you specify **const** when declaring an aggregate, all the aggregate members are treated as objects of type **const**.
- You can specify **const** with **volatile**, or any of the storage-class specifiers or modifiers.
- If you attempt to access a **const** object using a pointer to that object not declared **const**, the result is undefined.
- The address of a non-**const** object can be assigned to a pointer to a **const** object to a **const** pointer, but you cannot use that pointer to alter the value of the object. The result is undefined.

The following example declares the variable x to be a constant integer:

```
int const x;
```

When declaring pointers, depending upon the placement of the **const** modifier in the declaration, VAX C interprets either the pointer or the object to which it points as the constant variable. For instance, the following example declares the variable y to be a constant pointer to an integer because the **const** modifier appears after the asterisk:

```
int * const y;
```

In the following example, the variable z is declared as a pointer to a constant integer because the asterisk appears after the **const** modifier:

```
int const * z;
```

When you specify the **const** modifier in association with a **globaldef** specifier that identifies a psect, be aware that all variables declared have their storage allocated in the psect and that an inconsistent use of the **const** modifier can alter the psect attribute and lead to diagnostic messages. For detailed information about psects and the VAX C storage classes, see Chapter 14. For instance, the following examples are valid uses of the **const** modifiers. Specifically, in Example 1 the variable x becomes a nonconstant pointer to a constant integer that assigns the WRT attribute to the psect. In Example 2, the variable y becomes a constant pointer to an integer and assigns the NOWRT attribute to the psect. In Example 3, the variable z becomes a constant variable contained in the psect and assigns it the NOWRT attribute.

Example 1

```
globaldef {"psect"} const int * x;
```

Example 2

```
globaldef {"psect"} int * const y;
```

Example 3

```
globaldef {"psect"} const int z;
```

VAX C generates a warning message when there is an inconsistent usage of the **const** modifier, as shown in the following example:

```
globaldef {"psect"} const int test, * bar;
```

In this example, the variable `test` is declared as a constant variable that is allocated in the `psect` and assigns it the `NOWRT` attribute. The variable `bar` is a pointer that is not itself constant, but that points to a constant integer. In this case, VAX C automatically causes the pointer to become constant. DIGITAL recommends that you not mix constant and nonconstant variables in a **globaldef** declaration that names a `psect`, as your program may generate unpredictable results.

9.7.2 The volatile Modifier

The **volatile** data-type modifier prevents an object from being stored in a machine register, forcing it to be allocated in memory. This data-type modifier is useful for declaring data that is to be accessed asynchronously. A device driver application often uses **volatile** data storage.

The following rules apply to the use of the **volatile** modifier:

- You can specify **volatile** with any of the other data-type keywords in a declaration.
- If you specify **volatile** when declaring an aggregate, all the aggregate members are treated as objects of type **volatile**.
- You can specify **volatile** with **const**, or any of the storage-class specifiers or modifiers *except* the storage class **register**.
- The address of an object of some other type can be assigned to a **volatile** pointer, but the rules of the **volatile** data-type modifier must be followed if you refer to the object using that pointer.

9.8 Storage-Class Modifiers

The VAX C compiler can accept a storage-class specifier and a storage-class modifier in any order; usually, the modifier is placed after the specifier in the source code. For example:

```
extern noshare int x;
    /* Or, equivalently . . . */
int noshare extern x;
```

The following sections describe each of the VAX C specific storage-class modifiers in detail.

9.8.1 The noshare Modifier

The **noshare** storage-class modifier assigns the attribute NOSHR to the program section of the variable. Use this modifier to allow other programs, as shareable images, to have a copy of the variable's psect without the shareable image changing the variable's value in the original psect.

When a variable is declared with the **noshare** modifier and a shared image that has been linked to your program refers to that variable, a copy is made of the variable's original psect to a new psect in the other image. The other program may alter the value of that variable within the local psect without changing the value still stored in the psect of the original program.

For example, if you need to establish a set of data that will be used by several programs to initialize local data sets, then a VAX C program can do this by declaring the external variables using the **noshare** specifier. Each program receives a copy of the original data set to manipulate, but the original data set remains for the next program to use. If you define the data as [**extern**] without the **noshare** modifier, a copy of the psect of that variable is not made; each program would be allowed access to the original data set and the initial values would be lost. If the data is declared as **const** or **readonly**, each program is able to access the original data set, but none of the programs can then change the values.

You can use the **noshare** modifier with the storage-class specifiers **static**, [**extern**], **globaldef**, and **globaldef**{"name"}. For more information about the possible combinations of specifiers and modifiers, and the effects of the storage-class modifiers on program section attributes, see Chapter 14.

You can use **noshare** alone, which implies an external definition of storage class [**extern**]. Also, when declaring variables using the [**extern**] and **globaldef**{"name"} storage-class specifiers, you can use **noshare**, **const**, and **readonly**, together, in the declaration. If you declare variables using the **static** or the **globaldef** specifiers, and you use both of the modifiers in the declaration, the compiler ignores **noshare** and accepts **const** or **readonly**.

9.8.2 The `readonly` Modifier

The **`readonly`** storage-class modifier, like the data-type modifier **`const`**, assigns the NOWRT attribute to the variable's program section; if used with the **`static`** or **`globaldef`** specifier, the variable is stored in the psect \$CODE, which has the NOWRT attribute by default.

You can use both the **`readonly`** and **`const`** modifiers with the storage-class specifiers **`static`**, **`[extern]`**, **`globaldef`**, and **`globaldef`** {"psect"}.

In addition, both the **`readonly`** modifier and the **`const`** modifier can be used alone. When you specify these modifiers alone, an external definition of storage class **`[extern]`** is implied.

The **`const`** modifier restricts access to data in the same manner as the **`readonly`** modifier. However, in the declaration of a pointer, the **`readonly`** modifier cannot appear between the asterisk and the pointer variable to which it applies.

The following example shows the similarity between the **`const`** and **`readonly`** modifiers. In both instances, the variable `point` represents a constant pointer to a nonconstant integer.

```
readonly int * point;
int * const point;
```

NOTE

For new program development, DIGITAL recommends that you use the **`const`** modifier.

9.8.3 The `_align` Modifier

The **`_align`** storage-class modifier allows you to align objects of any of the VAX C data types on a specified storage boundary. Use the **`_align`** modifier in a data declaration or definition.

For example, if you want to align an integer on the next quadword boundary, you can use any of the following declarations:

```
int _align( QUADWORD ) data;
int _align( quadword ) data;
int _align( 3 ) data;
```


When specifying the boundary of the data alignment, you can either use a predefined constant or you can specify an integer value that is a power of 2. The power of 2 tells VAX C the number of bytes to pad in order to align the data. In the previous example, integer 3 specifies an alignment of 2^3 bytes, which is 8 bytes—a quadword of memory.

Table 9–5 presents all the predefined alignment constants, their equivalent power of 2, and their equivalent number of bytes.

Table 9–5: Predefined Alignment Constants

Constant	Power of 2	Number of Bytes
BYTE or byte	0	0
WORD or word	1	2
LONGWORD or longword	2	4
QUADWORD or quadword	3	8
OCTAWORD or octaword	4	16
PAGE or page	9	512

Preprocessor Directives

Preprocessor directives are lines in the source file that direct the compiler to alter its normal processing of VAX C source code. Preprocessor directives are not defined formally by the C language, so their implementation may vary from one compiler to another. For example, in most implementations of C running on UNIX systems, the preprocessor is a separate program that operates before the compiler, as the name preprocessor implies. In VAX C, these directives are executed in an early phase of the compiler.

If you plan to port programs to and from other C implementations, take care in choosing which preprocessor directives to use within your programs. See Section 10.3 for more information about conditional compilation. For a complete discussion of portability concerns, see the *VAX C Run-Time Library Reference Manual*.

This chapter discusses the following preprocessor directives:

- **#define** and **#undef**—Defines macro substitutions and replacements. (See Section 10.1.)
- **#dictionary**—Extracts Common Data Dictionary (CDD) data definitions and includes them in the source file. (See Section 10.2.)
- **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, **#endif**, and the **defined** operator—Controls under which conditions segments of code are to be compiled or not. (See Section 10.3.)
- **#include**—Includes source text from an external file or library. (See Section 10.4.)
- **#line**—Specifies a new line number and file name at the terminal, not in the listing file. (See Section 10.5.)
- **#module**—Specifies a module name to the VMS Linker. (See Section 10.6.)

- **#pragma**—Performs an implementation-specific task. (See Section 10.7.)

Preprocessor directives are independent of the usual scope rules; they remain in effect from their occurrence until the end of the compilation unit. For more information about the compilation unit, see Chapter 1.

10.1 Macro Definitions (**#define** and **#undef**)

The **#define** directive specifies a macro identifier and a token string. The token string is substituted for every subsequent occurrence of that identifier in the program text, unless it occurs inside a **char** constant, a comment, or a quoted string. You use the **#undef** directive to cancel a definition for a macro.

NOTE

Previous versions of this guide refer to these macros as tokens.

The syntax of the **#define** directive is as follows:

```
#define identifier token-string  
#define identifier(identifier, . . . ) token-string
```

If you omit the token-string, the identifier is deleted from the text to be processed by the compiler.

After a token string is substituted in the source file, the compiler rescans the source line from the beginning of the substituted text to determine whether the previously inserted text contains identifiers defined by other **#define** directives. If so, the identifiers are replaced by their currently specified token strings. Example 10–1 shows nested **#define** directives.

NOTE

/DEFINE and **/UNDEFINE** perform almost the same functions from the command line as **#define** and **#undefine**. See Section 1.3.2 for more information.

Example 10–1: Nested Substitution Directives

```
/* Show multiple substitutions and listing format.          */
#define AUTHOR james + LAST

main()
{
    int writer,james,michener,joyce;

#define LAST michener
    writer = AUTHOR;

#define LAST joyce
    writer = AUTHOR;
}
```

Compile Example 10–1 with the following command:

```
$ CC/LIST/SHOW=INTERMEDIATE EXAMPLE RETURN
```

The following listing results:

```
1          /* Show multiple substitutions and
2          listing format.          */
3          #define AUTHOR james + LAST
4
5          main()
6          {
7      1      int writer,james,michener,joyce;
8      1
9      1      #define LAST michener
10     1      writer = AUTHOR;
11           1      writer = james + LAST;
12           2      writer = james + michener;
13           1      #define LAST joyce
14           1      writer = AUTHOR;
15           1      writer = james + LAST;
16           2      writer = james + joyce;
17           1      }
```

On the first pass, the compiler replaces the identifier `AUTHOR` with the token string `james + LAST`. On the second pass, the compiler replaces the identifier `LAST` with its currently defined token string value. At line 9, the token string value for `LAST` is the identifier `michener`, so `michener` is substituted at line 10. At line 12, the token string value for `LAST` is redefined to be the identifier `joyce`, so `joyce` is substituted at line 13. The following line is the final text that the compiler processes:

```
writer = james + joyce;
```

The **#define** directive may be continued onto subsequent lines if necessary. To do this you must end each line to be continued with a backslash (\) immediately followed by a newline character. The backslash and newline do not become part of the definition. The first character in the next line is logically adjacent to the character that immediately precedes the backslash. The backslash/newline as a continuation sequence is valid anywhere. Comments within the definition line can be continued without the backslash/newline.

10.1.1 Constant Identifiers

The first form of the **#define** directive defines a simple substitution, usually of a constant for a frequently used identifier. The identifier can be 255 characters long and include the continuation character (\). A common use of the directive is to define the end-of-file (EOF) indicator as follows:

```
#define EOF (-1)
```

The substitution text for this example is delimited with parentheses to avoid lexical ambiguities when the text is substituted in the program.

For example:

```
i=EOF;
```

If the token string `-1` is substituted for the identifier `EOF`, then the contiguous characters (`==`) may be mistaken for an operator.

10.1.2 Canceling Definitions (**#undef**)

The syntax for the **#undef** directive is as follows:

```
#undef identifier
```

This directive cancels a previous definition of the identifier by **#define**.

10.1.3 Macro Parameters

Some macros include a list of parameters. These macro substitutions look like function calls. If you call a function, control passes from the program to the function object code (or, optionally, the function's shareable image) at run time; if you reference a macro, source code is inserted into the program at compile time. The parameters are replaced by the corresponding arguments and the text is inserted into the program stream. The syntax of a macro definition is as follows:

```
#define name([parm1[,parm2, ...]]) [token-string]
```

The name, parm1, parm2, and so forth are identifiers, and the token-string is arbitrary text.

After the macro definition, all macro references in the source code with the following form are replaced by the token string from the directive and any formal parameters that appear in the token string are replaced by the corresponding arguments from the reference. For example, argument arg1 replaces parameter parm1, and so forth.

```
name([arg1[,arg2, ... ]])
```

As shown in the syntax of the macro definition, the token string is optional. If the token string is omitted from the macro definition, the entire macro reference disappears from the source text.

The token string in the macro definition, as well as arguments in a macro reference, may contain other macro references. Substitution occurs, but such nested references are limited to a depth of 64. The maximum number of parameters or arguments is also 64.

The VAX C RTL macro **_toupper** is a good example of macro substitution. This macro is defined in the *ctype* definition module in the following manner:

```
#define _toupper(c) ((c) >= 'a' && (c) <= 'z' ? (c) & 0X5F : (c))
```

When you reference the macro **_toupper**, the compiler replaces the macro keyword and its parameter with the token string from the directive. The token string of VAX C source code looks cryptic but can be translated in the following manner: if parameter *c* is a lowercase letter (between 'a' and 'z'), the expression evaluates to an uppercase letter ((*c*) & 0X5F); otherwise, it evaluates to the character as given. This token string uses the if-then-else conditional operator (? :). For more information about the conditional operator, see Section 7.6. For more information about the bitwise operators, see Section 7.5.5.

Preprocessor directives and the macro references have syntax that is independent of the VAX C language. The following list gives the rules for specifying macro definitions:

- The macro name and the formal parameters are identifiers and are specified according to the rules for identifiers in the VAX C language.
- Spaces, tabs, and comments may be used freely within a **#define** directive. In particular, they may appear anywhere that the delta symbol (Δ) appears in the following example:

```
# $\Delta$  define  $\Delta$  name ( $\Delta$  parm1 $\Delta$  , $\Delta$  parm2 $\Delta$  ) $\Delta$  \  
  
 $\Delta$  macro-string $\Delta$ 
```

- White space cannot appear between the name and the left parenthesis that introduces the parameter list. White space may appear inside the token string. Also, at least one space, tab, or comment must separate name from **define**. Comments may appear within the token string, but they do not become part of the macro definition.

The following list gives the rules for specifying macro references:

- Comments and white space characters (spaces, horizontal and vertical tabs, carriage returns, newlines, and form feeds) may be used freely within a macro reference. In particular, they may appear anywhere that the delta symbol (Δ) appears in the following example:

```
 $\Delta$  name $\Delta$  ( $\Delta$  arg1 $\Delta$  , $\Delta$  arg2 $\Delta$  )
```

- Arguments consist of arbitrary text. Syntactically, they are not restricted to VAX C expressions. They may contain embedded comments and white space. Comments are ignored, but the white space is preserved during the substitution.
- The number of arguments in the reference must match the number of parameters in the macro definition, but individual arguments may be null.
- Commas separate arguments except where they occur inside string or character constants, comments, or parentheses. You must balance parentheses within arguments.

Take care when specifying the token string. Since the token string consists of arbitrary text, the replacement of parameters with arguments occurs even if a parameter appears inside a character or string constant within the token string. To be recognized, you should delimit a parameter from the surrounding text by white space or punctuation characters, such as parentheses.

You must be careful when specifying macro arguments that use the increment ($++$), decrement ($--$), and assignment (such as $+=$) operators or other arguments that may cause side effects. Function calls are another source of possible side effects. For example, do not pass the following argument to the **_toupper** macro:

```
_toupper(p++)
```

When the argument $p++$ is substituted in the macro definition, the effect within the program stream is as follows:

```
((p++) >= 'a' && (p++) <= 'z' ? (p++) & 0X5F : (p++))
```

At run time, these expressions may not be evaluated in left-to-right order. For this reason, specifying macro arguments that may cause side effects is not good programming practice. Even if you are aware of possible side effects, the token strings within macro definitions may be changed, which changes the side effects without warning.

NOTE

If Version 3.4 or earlier of the VAX Common Data Dictionary is installed on your system, references in this manual to the “VAX Common Data Dictionary,” “Common Data Dictionary,” or “CDD” refer to the VAX common Data Dictionary installed on your system.

If the VAX CDD/Plus Version 3.4 is installed on your system, references in this manual to the “VAX Common Data Dictionary,” “Common Data Dictionary,” or “CDD” refer to the DMU format dictionary. CDD/Plus supports dictionary definitions in two distinct formats:

- DMU format—dictionary definitions that can be created and manipulated with the DMU, CDDL, and CDDV utilities, and other products that do not support the new features of CDD/Plus.
- CDO format—dictionary definitions that can be created and manipulated with the CDO utility, the CDD/Plus call interface, and other supporting products.

Definitions that you intend to use in VAX C programs can be created and manipulated in the DMU format dictionary using DMU, CDDL, CDDV, and other products that support DMU dictionary definitions.

Your site may have other products that support the new features of CDD/Plus. If so, you may benefit by using CDO, the CDD/Plus call interface, or other supporting products to create definitions in the CDO format dictionary. The CDO dictionary definitions you create can be used by both your VAX C programs and the products that support the new features of CDD/Plus.

For more information on the DMU format dictionary, CDO format dictionaries, and CDD/Plus in general, see the *VAX CDD/Plus User's Guide*.

10.1.4 Listing Substituted Lines

The `/SHOW` command-line qualifier has two optional values that enable the listing of all lines modified by macro substitutions. The values are `EXPANSION` and `INTERMEDIATE`.

Consider the following qualifiers:

```
/LIST/SHOW=EXPANSION
```

The listing produced by the compiler with these qualifiers shows both the original line and the final form of the substituted line. Substituted lines are flagged in the margin with numbers designating the nesting level of substitution.

Consider the following qualifiers:

```
/LIST/SHOW=INTERMEDIATE
```

The compiler lists all intermediate substitutions with one substitution per line.

Without one of these two qualifiers or `/SHOW=ALL`, the compiler only lists the original form of a line.

Example 10–1 shows the effect of the `/SHOW=INTERMEDIATE` qualifier. For more information about the format of VAX C compiler listings, see Chapter 1.

10.2 Common Data Dictionary Extraction (#dictionary)

The Common Data Dictionary (CDD) is an optional VMS software product, available under a separate license, that maintains a set of data structure definitions that many programs on a system can access. These data definitions are written in a language-independent form and are translated into the target language when they are included in the program stream.

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

The CDD features the following advantages:

- Record declarations are language-independent and can be shared across VAX languages that support the CDD.

- Data definitions are centrally located, which helps reduce the amount of duplicated effort in a programming project.
- A single declaration helps guarantee the accuracy and reliability of data.

For detailed information about the CDD, see the *VAX Common Data Dictionary Reference Manual*, the *VAX Common Data Dictionary Utilities Manual*, and the *VAX Common Data Dictionary Language Reference Manual*.

10.2.1 Using the #dictionary Directive

The **#dictionary** preprocessor directive is VAX C specific, and allows you to extract CDD data definitions and include these definitions in your program. The format of the **#dictionary** directive is as follows:

```
#dictionary cdd_path
```

The `cdd_path` is a character string that gives the path name of a CDD record, or a macro that expands to the path name of the record. For example:

```
#dictionary "CDD$TOP.personnel.service.salary_record"
```

This path name describes all subdirectories leading to the `salary_record` data definition, beginning with the root directory (`CDD$TOP`).

You can use the logical name `CDD$DEFAULT` to define a default path name for a dictionary directory. This logical name can specify part of the path name for the dictionary object. For example, you can define `CDD$DEFAULT` as follows:

```
$ DEFINE CDD$DEFAULT CDD$TOP.PERSONNEL
```

When this definition is in effect, the **#dictionary** directive can contain the following:

```
#dictionary "service.salary_record"
```

CDD definitions are written in the Common Data Dictionary Language (CDDL), and are included in a dictionary with the `CDDL` command. For example, you can write a definition for a structure containing someone's first and last name as follows:

```

define record cdd$top.doc.cname_record.
  cname structure.
    first   datatype is text
           size is 20 characters.
    last    datatype is text
           size is 20 characters.
  end cname structure.
end cname_record record.

```

If this definition is found in a source file named CNAME.DDL, it can be included in the CDD subdirectory named doc by entering the following command:

```
$ CDDL cname
```

After executing this command, a VAX C program can reference this definition with the **#dictionary** directive. If the **#dictionary** directive is not embedded in a VAX C structure declaration, then the resulting structure is declared with a tag name corresponding to the name of the CDD record. Consider the following example:

```
#dictionary "cdd$top.doc.cname_record"
```

This line of VAX C code results in the following declarations:

```

struct cname
{
  char first [20];
  char last  [20];
};

```

You can embed the **#dictionary** directive in another VAX C structure declaration as follows:

```

struct
{
  int id;
#dictionary "cname_record"
} customer;

```

These lines result in the following declaration, which uses `cname` as an identifier for the embedded structure:

```

struct
{
  int id;
  struct
  {
    char first [20];
    char last  [20];
  } cname;
} customer;

```

If you specify `/LIST` and either `/SHOW=DICTIONARY` or `/SHOW=ALL` in the compilation command line, then the translation into VAX C of the CDD record description is included in the listing file and marked with the letter `D` in the margin.

10.2.2 Support for CDD Data Types

The CDD supports all VMS data types. VAX C can translate all the VMS data types when they are declared in CDD records. Data types that do not occur naturally in the VAX C language are handled as follows:

- VAX C never attempts to approximate a data type that is not supported by the C language.
- Instead of approximating a data type, VAX C uses its own structure data type to represent all types not supported by the VAX C language (except for excessively long bit strings); specifically, VAX C creates structures of arrays of type `char` that are large enough to represent the data structure.
- Bit strings (aligned or unaligned) may be up to 32 bits long, as defined by the VAX C language. Bit strings longer than 32 bits are broken into increments of 32-bit strings or smaller so that the structure is correct with respect to size. However, the long bit string cannot be accessed as one unit.
- All row-major arrays are represented as zero-origin arrays of the appropriate size. An informational message is issued if the record description specifies nonzero-origin dimension bounds. The compiler adjusts the upper bound appropriately to maintain the correct number of elements relative to a lower bound of zero. Column-major arrays are converted to one-dimensional arrays containing the same total number of elements.

The compiler applies various consistency checks to the record attributes extracted from the CDD, particularly the field data-type attributes. An error message is issued when a record description does not pass the consistency checks. An informational message is issued when VAX C is confronted with facility-independent attributes that are not supported. An error message is issued when an attribute that is required by VAX C is not present, even if the attribute is optional in the CDD record protocol.

The compiler synthesizes names for unnamed and filler fields. If the CDD does not specify a name and a name is required by the syntax of the VAX C language, the compiler synthesizes the name `cc_cdd$_unnamed_nnnnn`. When the CDD specifies a filler or a name that VAX C does not support, the

compiler synthesizes the name `cc_cdd$_filler_#nnnnn`, which includes the pound sign character (`#`). The string `nnnnn` represents a unique integer. The `#` is not a valid character in an identifier, so you cannot reference these fields.

Unsupported data types are mapped into VAX C as structures of character arrays of the appropriate size. The declaration of these data types uses the following format:

```
struct { char Cname [s]; } CDDname;
```

The `CDDname` is the name of the member in the CDD record. `Cname` is an identifier of the form `cc_cdd$_unsupported_#nnnnn`, where `nnnnn` is a unique integer, and `s` is the size of the data item, in bytes.

VAX C generates `variant_struct` or `variant_union` declarations for unnamed CDD structures and unions, so you do not have to specify these references.

Table 10–1 summarizes the mapping between CDD data types and VAX C data types.

Table 10–1: Mapping Between CDD and VAX C Data Types

CDD Data Type	C Data Type
Unspecified	Unsupported
Unsigned byte	unsigned char
Unsigned word	unsigned short
Unsigned longword	unsigned int
Unsigned quadword	Unsupported
Unsigned octaword	Unsupported
Signed byte	char
Signed word	short
Signed longword	int
Signed quadword	Unsupported
Signed octaword	Unsupported

(continued on next page)

Table 10–1 (Cont.): Mapping Between CDD and VAX C Data Types

CDD Data Type	C Data Type
F_floating	float
D_floating	double ¹
G_floating	double ¹
H_floating	Unsupported
F_floating complex	Unsupported
D_floating complex	Unsupported
G_floating complex	Unsupported
H_floating complex	Unsupported
Text	char [<i>n</i>]
Varying text ²	Unsupported
Numeric string:	
Unsigned	Unsupported
Left separate	Unsupported
Left overpunch	Unsupported
Right separate	Unsupported
Right overpunch	Unsupported
Zoned sign	Unsupported
Packed decimal string	Unsupported
Bit	Bit field ³
Bit unaligned	Bit field ³
Date and time	Unsupported
Date	Unsupported
Virtual field	Ignored
Varying string ²	Unsupported

¹A message may be issued depending upon the specification of the /G_FLOAT qualifier. If the data type of the CDD record member is D_floating and the /G_FLOAT command qualifier was specified, or if the data type of the record member is G_floating and the /NOG_FLOAT command qualifier was specified, an informational message is issued and the member is represented as **struct { char [8]}** instead of **double**.

²For these data types, the length of the structure is two bytes longer than the string to allow for the length field.

³A message is issued if the bit string length is greater than 32.

10.3 Conditional Compilation (**#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, and **#endif**)

Six directives are available to control conditional compilation. They delimit blocks of statements that are compiled if a certain condition is true. You can

nest these directives. The beginning of the block of statements is marked by one of three directives: **#if**, **#ifdef**, or **#ifndef**. Optionally, an alternative block of statements can be set aside with the **#else** or the **#elif** directives. The end of the block is marked by an **#endif** directive.

If the condition checked by **#if**, **#ifdef**, or **#ifndef** is true, then VAX C ignores all lines between an **#else** (or **#elif**) and an **#endif** directive.

If the condition is false, then the lines between the **#if**, **#ifdef**, or **#ifndef** and an **#else**, (or **#elif**) or **#endif** directive are ignored. The compiler flags ignored lines with the letter X in the compiler listing margin.

The **#if** directive has the following form:

```
#if constant-expression
```

This directive checks whether the constant expression is nonzero (true). The operands must be constants. The increment (**++**), decrement (**--**), **sizeof**, pointer (*****), address (**&**), and cast operators are not allowed in the constant expression.

The constant expression in an **#if** directive is subject to text replacement and can contain references to identifiers defined in previous **#define** directives. The replacement occurs before the expression is evaluated.

If an identifier used in the expression is not currently defined, the compiler treats the identifier as though it were the constant zero. The compiler issues a diagnostic message for this if **/STANDARD=PORTABLE** is specified on the compilation.

The **#ifdef** directive has the following form:

```
#ifdef identifier
```

This directive checks whether the identifier was previously defined by a **#define** directive.

The **#ifndef** directive has the following form:

```
#ifndef identifier
```

This directive checks to see if the identifier is not defined or if it has been undefined by the **#undef** directive.

The **#else** directive has the following form:

```
#else
```

This directive delimits alternative source lines to be compiled if the condition tested for in the corresponding **#if**, **#ifdef**, or **#ifndef** directive is false. An **#else** directive is optional.

The **#elif** directive has the following form:

```
#elif constant-expression
```

The **#elif** directive performs a task similar to the combined use of the **else-if** statements in VAX C. This directive delimits alternative source lines to be compiled if the constant expression in the corresponding **#if**, **#ifdef**, or **#ifndef** directive is false *and* if the additional constant expression presented in the **#elif** line is true. An **#elif** directive is optional.

The **#endif** directive has the following form:

```
#endif
```

This directive ends the scope of the directives.

The number of **#endif** directives necessary changes according to whether the **#elif** or **#else** directive is used. Consider the following equivalent examples:

```
#if true
.
.
.
#elif false
.
.
.
#endif

#if true
#else
  #if
.
.
.
#endif
#endif
```

10.3.1 The defined Operator

If you need to check to see if many macros are defined, you may use the special **defined** operator in a single use of the **#if** directive. In this way, you can check for macro definitions in one concise line without having to use many **#ifdef** or **#ifndef** directives.

For example, suppose you want to check the following macros:


```

#ifdef macro1
printf( "Oh, Mary!\n" )
#endif

#ifndef macro2
printf( "Oh, Mary!\n" )
#endif

#ifdef macro3
printf( "Oh, Mary!\n" )
#endif

```

You can use the **defined** operator in a single use of the **#if** preprocessor directive, as follows:

```

#ifdef (macro1) || !defined (macro2) || defined (macro3)
printf( "Oh, Mary!\n" )
#endif

```

You can use **defined** like any other operator. However, you can only use **defined** in the evaluated expression of an **#if** or **#elif** preprocessor directive.

10.4 File Inclusion (**#include**)

The **#include** directive inserts external text into the macro stream delivered to the compiler. Often, global definitions for use with VAX C functions and macros are included in the program stream with the **#include** directive. The **#include** directives may be nested to a depth determined by the FILLM process quota and by virtual memory restrictions. The VAX C compiler imposes no inherent limitation on the nesting level of inclusion.

In VAX C source programs, the inclusion of both VMS and most DEC/Shell file specifications are legal. An example of a valid DEC/Shell file specification is as follows:

```
BEATLE!/DBAO/MCCARTNEY/SONGS.LIS.3
```

The exclamation point (!) separates the node name from the rest of the specification; slash characters (/) separate devices and directories; periods (.) separate file extensions and file versions. Since one character is used to separate two segments of the file specification, ambiguity can occur. For more information on including DEC/Shell file specifications, see the *VAX C Run-Time Library Reference Manual*.

The following sections describe the forms of the **#include** directive.

10.4.1 Inclusion Using Angle Brackets

The first form of the directive is as follows:

```
#include <file-spec>
```

The identifier `file-spec` is a valid file specification or a logical name. A file specification may be up to 255 characters long. The compiler first translates the specified file name to see if it is a valid VMS specification. If it is not, the compiler then checks to see if it is a valid DEC/Shell specification; if it is, it translates the specification to a valid VMS specification using VAX C RTL functions. If the specification is not valid for either VMS or the DEC/Shell specification, an error occurs. Valid DEC/Shell file specifications are a subset of valid UNIX file specifications. For more information about valid DEC/Shell file specifications, see Chapter 1.

When specifying the names of files to be included in your source program, avoid directory specifications of the following form:

```
DBA0:[dir-name . . . ]
```

Depending on the location of your program source file, and your current RMS default directory, this form of directory specification may or may not translate to the intended directory. When specifying files and their directories, use complete directory specifications.

This form of file inclusion delimits the file specification with angle brackets (<>). If VAX C encounters this form of file inclusion, the compiler searches directories in the following order:

1. The directories specified in the `/INCLUDE_DIRECTORY` qualifier, if `/INCLUDE_DIRECTORY` was used (see Section 1.3.2)
2. The directory or search list of directories specified in the logical name `VAXC$INCLUDE`, if `VAXC$INCLUDE` is defined
3. If `VAXC$INCLUDE` is not defined, then the directory or search list of directories specified by `SYS$LIBRARY`

You can define `VAXC$INCLUDE` to be a valid directory specification or a search list of valid directory specifications. Before each compilation of your program, you have the flexibility of redefining `VAXC$INCLUDE` to be any valid directory or list of directories you choose.

You cannot define `VAXC$INCLUDE` to be a rooted directory or subdirectory of the following form:

```
DBA0:[dir-name.]
```

When defining `VAXC$INCLUDE`, use complete directory specifications.

If `VAXC$INCLUDE` translates to a directory or a search list of directories, and if the compiler cannot locate the specified file, the compiler generates an error message. If `VAXC$INCLUDE` is undefined, the compiler then searches the directory `SYS$LIBRARY` for the specified file; if the file cannot be found, the compiler generates an error message. For more information about search lists, see the DCL command `DEFINE` in the *VMS DCL Dictionary*.

When porting programs to the VMS environment, your programs may contain **#include** directives of the following form:

```
#include <sys/file.h>
```

The VAX C compiler translates this line, common in programs that run on UNIX systems, to the following DEC/Shell path name:

```
/sys/file.h
```

The compiler then translates the DEC/Shell path name to the VMS file specification as follows:

```
SYS:FILE.H
```

If you port programs containing such directives, define the logical `SYS` to be the proper name of the VMS directory containing the files to be included.

10.4.2 Inclusion Using Quotation Marks (" ")

The second form of the **#include** preprocessor directive is as follows:

```
#include "file-spec"
```

The identifier `file-spec` is a valid VMS or DEC/Shell file specification.

This form of file inclusion delimits the file specification with quotation marks (" "). The compiler searches directories in the following order:

1. The directory containing the top-level source file
2. The directories specified in the `/INCLUDE_DIRECTORY` qualifier, if `/INCLUDE_DIRECTORY` was specified
3. The directory or search list of directories specified in the logical name `C$INCLUDE`, if `C$INCLUDE` is defined

The top-level, source-file directory contains the compiled source file for the included file and is not necessarily the current RMS default directory. For VAX C on VMS systems, the directory containing the source file (.C file) is the top-level source directory. On some other operating systems, the source directory is determined by the immediately nesting file, whether the nesting file is an include file or a .C file.

For example, given the current directory, DBA0:[CURRENT], and the following CC command line, the compiler searches DBA0:[OTHERDIR] for any included files delimited by quotation marks, even though the current RMS default is the directory, DBA0:[CURRENT]:

```
$ CC DBA0:[OTHERDIR]EXAMPLE.C[RETURN]
```

If the compiler cannot locate the specified file, it searches any directories specified by the /INCLUDE_DIRECTORY qualifier.

If the compiler still cannot locate the specified file, it translates the logical name C\$INCLUDE. If C\$INCLUDE translates to a valid directory specification or a search list of directories, the compiler searches that directory or directories for the specified file. Before each compilation of your program, you have the flexibility of redefining C\$INCLUDE to be any valid directory or list of directories you choose.

As with VAXC\$INCLUDE, do not define C\$INCLUDE to be a rooted directory or subdirectory. Use complete directory specifications when defining C\$INCLUDE.

If you defined C\$INCLUDE, and the compiler cannot locate the specified file in that directory or search list of directories, the compiler generates an error. If C\$INCLUDE is undefined, the search for the specified file ends in the directory containing the source file; the compiler searches no other directories. For more information about search lists, see the DCL command DEFINE in the *VMS DCL Dictionary*.

NOTE

If you include a file from SYS\$LIBRARY by using the angle brackets, and if the included file contains a second **#include** line that delimits the file specification with quotation marks, the compiler searches the directory containing the source file for the specified file, not SYS\$LIBRARY. When nesting **#include** directives as previously described, the file specification in quotation marks must contain complete device and directory information.

10.4.3 Inclusion of Text Modules

The third form of the **#include** preprocessor directive is as follows:

```
#include module-name
```

The identifier `module-name` is the name of a module in a text library. This method of inclusion is the most efficient on VMS, because modules within a text library are indexed and easier to manipulate than files in a directory. However, this format is VAX C specific and is not portable.

VAX C text libraries are specified and searched as follows:

- A text library can be created with the `LIBRARY` command and specified with the `/LIBRARY` qualifier on the `CC` command line.
- If you compile more than one compilation unit using a single `CC` command, you must specify the library within each of the compilation units, if needed. Consider the following example:

```
$ CC sourcea+mylib/LIBRARY, sourceb+mylib/LIBRARY
```

- If you specify more than one library to the VAX C compiler, and if the `#include` directives are not nested (see the note in the previous section), then the libraries are searched in the specified order each time an `#include` directive is encountered. Consider the following example:

```
$ CC sourcea+mylib/LIBRARY+yourlib/LIBRARY
```

In this example, the compiler searches for modules referenced in `#include` directives first in `MYLIB.TLB` and then in `YOURLIB.TLB`.

- If no library is specified on the `CC` command line, or if the specified module cannot be found in any of the specified libraries, the following actions are taken:
 - If you defined an equivalence name for `C$LIBRARY` that names a text library, that library is searched.
 - The compiler searches for any remaining unresolved module names in `SYS$LIBRARY:VAXCDEF.TLB`.

10.4.4 Macro Substitution in `#include` Directives

VAX C allows macro substitution within the `#include` preprocessor directive.

For instance, if you want to include a file name, you can use the following two directives:

```
#define macrol "file.ext"  
#include macrol
```

If you use defined macros in **#include** directives, the macros must evaluate to one of the three following acceptable **#include** file specifications or the use generates an error message:

```
<file-spec>  
"file-spec"  
module-name
```

10.5 Specifying Line Numbers (**#line** and **#**)

The VAX C compiler keeps track of information about relative line numbers in each file involved in the compilation, and uses the number when it delivers diagnostic messages to the terminal. The compiler increments the subsequent lines from the line number specified by the **#line** directive. The directive can also specify a new file specification for the program source file. The **#line** directive does not change the line numbers in your compilation listing, only the line numbers given in diagnostic messages sent to the terminal screen. This directive is useful for locating errors in text that is included using the **#include** preprocessor directive.

The formats of the **#line** directive are as follows:

```
#line constant identifier  
#line constant string  
# constant identifier  
# constant string
```

The compiler gives the line following a **#line** directive the number specified by the parameter constant. The second parameter can be specified as either a VAX C identifier or a character-string constant. It supplies the valid VMS file specifications. The character string must not exceed 255 characters.

10.6 Specifying the Module Name and Identification (**#module**)

When you compile source files to create an object file, the compiler assigns to that file the first file name of those specified in the compilation unit. The compiler adds the .OBJ file extension to the object file. Internally, the VMS system (the debugger and the librarian) recognizes the object module by the file name; the compiler also gives the module a Version 1.0 version identification. For example, given the object file EXAMPLE.OBJ, the debugger recognizes the EXAMPLE object module. To change the

system-recognized module name and version number, use the **#module** directive.

You can find the module name and the module version number listed in the compiler listing file and the linker load map.

The syntax of the **#module** directive is as follows:

```
#module identifier identifier  
#module identifier string
```

The first parameter must be a valid VAX C identifier. It specifies the module name to be used by the linker. The second parameter specifies the optional identification that appears on listings and in the object file. It must be either a valid VAX C identifier of 31 characters or less, or a character-string constant of 31 characters or less.

Only one **#module** line can be processed per compilation unit, and that line must appear before any VAX C language text; it can follow other directives, such as **#define**, but it must precede any function definitions or external data definitions.

The parameters in a **#module** line are subject to text replacement and can, therefore, contain references to identifiers defined in previous **#define** directives. The replacement occurs before the parameters are processed.

The **#module** directive is VAX C specific and is not portable.

10.7 Implementation-Specific Preprocessor Directive (**#pragma**)

This section describes the implementation-specific preprocessor directives, or pragmas, that are available in the VAX C compiler. The **#pragma** directive is a standard method for implementing features that vary from one compiler to the next.

Note that **#pragma** directives are subject to macro expansion. A macro reference can occur anywhere after the keyword **pragma**. The following example demonstrates this feature using the **#pragma inline** directive:

```
#define opt inline  
#define f func  
#pragma opt(f)
```

The **#pragma** directive becomes **#pragma inline (func)** after both macros are expanded.

The following sections describe the **#pragma** directives.

10.7.1 #pragma [no]builtins Directive

The **#pragma [no]builtins** directive disables or provides access to the VAX C predefined functions. These functions do not result in a reference to a function in the run-time library or in your program. Instead, the compiler generates the machine instructions necessary to carry out the function directly at the call site. (For information on available built-in functions, see Section 11.2.)

The **#pragma [no]builtins** directive has the following format:

```
#pragma builtins
#pragma nobuiltins
```

10.7.2 #pragma ignore_dependency Directive

The **#pragma ignore_dependency** directive tells the compiler that the specified variables within the next encountered **for** or **while** loop should not inhibit the decomposition of that loop. This pragma affects only the next **for** or **while** loop encountered and does not affect nested loops.

The **#pragma ignore_dependency** directive has the following format:

```
#pragma ignore_dependency(id, . . . )
```

id

Is a pointer variable identifier that must be previously declared. If you specify only one identifier, you can omit the parentheses.

The **ignore_dependency** pragma must be placed after the declarations of all the variables specified to the pragma. In this way, VAX C checks to make sure that only variable names are specified to **#pragma ignore_dependency**.

When using this pragma, the generated code produces different results running in parallel than it does running sequentially (and also produces different results each time the code executes in parallel). However, some algorithms depend on convergence. For loops that use this algorithm, this pragma may be useful.

For more information about using decomposition pragmas, see Section 3.7.

10.7.3 #pragma [no]inline

The preprocessor directive **#pragma inline** suggests to the compiler that it provide inline expansion of the specified functions. Inline expansion of functions reduces execution time by replacing the function call with code that performs the actions of the original function code.

By default, VAX C attempts to provide inline expansion for all functions. The compiler also uses the following function characteristics to determine if it can provide inline expansion:

- Size
- Number of times the function is called
- Absence of the restrictions described in Section 10.7.3.1

The **#pragma inline** directive requests that the compiler attempt to provide inline code regardless of the size or number of times the specified functions are called. Functions that contain one of the restrictions described in Section 10.7.3.1 are never expanded inline, regardless of the use of the **#pragma inline**.

The **#pragma inline** directive has the following format:

```
#pragma inline (id, ...)
```

id

Is a C function identifier.

For instance, the following example specifies that the functions `push` and `pop` be expanded inline throughout the module in which the **#pragma inline** appears:

```
void push( int );
int pop(void);

#pragma inline( push, pop)

int stack[100];
int *stackp = &stack;

void push(int x)
{
    if (stackp == &stack)
        *stackp = x;
    else
        *stackp++ = x;
}

int pop()
{
    return *stackp--;
}
```

```

main()
{
    push(1);
    printf("The top of stack is now %d \n",pop());
}

```

The `/OPTIMIZE=NOINLINE` and the `/NOOPTIMIZE` qualifiers disable all **#pragma inline** directives that appear in your source code.

The **#pragma noline** can be used selectively to identify functions that are not to be expanded inline, even when the `/OPTIMIZE=INLINE` qualifier is used on the CC command line. The **#pragma noline** directive has the following format:

```
#pragma noline (id, ... )
```

id

Is a C function identifier.

10.7.3.1 Restrictions on Inline Expansion

If a function is to be expanded inline, you must place the function definition in the same module as the function call. The definition can appear either before or after the function call.

Functions cannot be expanded inline if they perform the following tasks:

- Take the address of an argument.
- Use an index expression that is not a compile-time constant in an array that is a field of a **struct** argument. An argument that is a pointer to a **struct** is not restricted.
- Use the *varargs* or *stdarg* package to access the function's arguments because they require arguments to be in adjacent memory locations, and inline expansion may violate that requirement.
- Declare an exception handler.

10.7.4 #pragma [no]member_alignment

By default, VAX C does not align structure members; they are stored on byte boundaries (with the exception of bit field members). However, you can use **#pragma member_alignment** to explicitly specify member alignment.

The **#pragma member_alignment** directive has the following format:

```
#pragma member_alignment
```

When **#pragma member_alignment** is used, the compiler aligns structure members on the next boundary appropriate to the type of the member, rather than on the next byte. For instance, a **long** variable is aligned on the next longword boundary; a **short** variable is aligned on the next word boundary.

Consider the following example:

```
#pragma nomember_alignment

struct x {
    char c;
    int b;
};

#pragma member_alignment

struct y {
    char c;          /*3 bytes of filler follow c */
    int b;
};

main ()
{
    printf( "The sizeof y is: %d\n", sizeof (struct y) );
    printf( "The sizeof x is: %d\n", sizeof (struct x) );
}
```

When this example is executed, it shows the difference between **#pragma member_alignment** and the directive **#pragma nomember_alignment**. The difference can also be seen by compiling with the **CC/LIS/SHOW=SYMBOLS** command and comparing the listed information for the two structures.

Once used, the **member_alignment** pragma remains in effect until the **nomember_alignment** pragma is encountered.

10.7.5 #pragma safe_call Directive

The **#pragma safe_call** directive tells the compiler that the specified function call or calls do not introduce data dependencies that prevent decomposition of a **for** or **while** loop during parallel processing. If you specify this directive outside of a function body, the functions named in the directive are globally safe in all **for** or **while** loops from the occurrence of the directive to the end of the compilation unit. If you specify this directive inside a function body, the functions named are safe only for the next **for** or **while** loop encountered in the enclosing function, and the pragma does not affect any **for** or **while** loops nested in the encountered loop.

The **#pragma safe_call** directive has the following format:

```
#pragma safe_call (id, . . . )
```

id

Is a function or pointer to a function that must be previously declared. If you specify only one identifier, you can omit the parentheses.

The **safe_call** pragma must be placed after the declarations of all functions specified to the pragma. In this way, VAX C checks to make sure that only function names are specified to **#pragma safe_call**.

Do not specify a function in a **safe_call** pragma if the function does the following:

- It has side effects that introduce data dependencies
- It is not reentrant
- It uses the VAX C Run-Time Library (RTL) routine **longjmp**, or otherwise modifies the normal flow of control
- It changes the process in some way
- It takes an address as an argument, and the address points to memory that is not shared

For more information about using decomposition pragmas, see Section 3.7.

10.7.6 #pragma sequential_loop Directive

The **#pragma sequential_loop** directive suppresses all decomposition analysis and prevents most decomposition diagnostics from being generated for the next **for** or **while** loop encountered in the program. This pragma affects only the next encountered **for** or **while** loop, and the pragma does not affect any nested **for** or **while** loops in the encountered loop.

The **#pragma sequential_loop** directive has the following format:

```
#pragma sequential_loop
```

For more information about using decomposition pragmas, see Section 3.7.

10.7.7 #pragma [no]standard Directive

Use **#pragma nostandard** to tell VAX C to ignore the current setting of the CC command-line qualifier `/STANDARD=PORTABLE` until further notice. It has no effect if the qualifier is not specified.

The **#pragma nostandard** directive has the following format:

```
#pragma nostandard
```

Use **#pragma standard** to tell VAX C to reinstate the setting of the `/STANDARD=PORTABLE` qualifier, but only if that qualifier is specified on the CC command line. It does not turn on portability checking if the `/STANDARD=PORTABLE` qualifier is not specified.

The **#pragma standard** directive has the following format:

```
#pragma standard
```

The **nostandard** and **standard** pragmas are together to define regions of source code where portability diagnostics are never to be issued. The following example demonstrates the use of these pragmas:

```
#pragma nostandard
extern noshare FILE *stdin, *stdout, *stderr;
#pragma standard
```

In this example, **nostandard** prevents the `NONPORTCLASS` diagnostic from being issued against the **noshare** storage-class modifier, which is VAX C specific.

Predefined Macros and Built-In Functions

This chapter describes the following topics:

- Predefined macros (Section 11.1)
- Built-in functions (Section 11.2)

VAX C predefines these macros and functions for your programming convenience. The macros assist in transporting code and performing simple tasks that are common to many programs. The built-in functions access VAX instructions very efficiently.

11.1 Predefined Macros

The following sections describe the VAX C predefined macros for use in your programs.

11.1.1 CC\$gfloat (G_Floating Identification Macro)

VAX C automatically defines a macro that can be used to identify whether you are compiling your program using the `G_floating` option. This macro can assist in writing code that executes conditionally, depending on whether the program is running using `D_floating` or `G_floating` precision.

If you used the `/G_FLOAT` qualifier, this symbol is defined as if the following were included before every compilation source group:

```
#define CC$gfloat 1
```

If you did not use the `/G_FLOAT` qualifier, this symbol is defined as if the following were included before every compilation source group:

```
#define CC$gfloat 0
```

You can conditionally assign values to variables of type **double** without causing an error and without being certain of how much storage was allocated for the variable. For example, external variables may be assigned values as follows:

```
#if CC$gfloat
double x = 0.12e308;          /* Range to 10 to the 308th power */
#else
double x = 0.12e38;          /* Range to 10 to the 38th power */
#endif
```

The VAX C compiler determines whether or not to substitute the value 1 for every occurrence of the predefined identifiers in a program; these identifiers are reserved by DIGITAL. The effect of these definitions may be removed by explicitly undefining the conflicting name. See Section 10.1.2 for more information about undefining. For more information about the G_floating representation of the **double** data type, see Chapter 8.

11.1.2 CC\$parallel (Parallel-Processing Identification Macro)

The VAX C compiler defines a macro that can be used to identify whether you are compiling your program using the /PARALLEL qualifier. This macro can assist in writing code that executes conditionally, depending on whether the program is running with or without parallel processing.

If you use the /PARALLEL qualifier, VAX C predefines CC\$parallel as if the following line appeared at the top of the compilation unit:

```
#define CC$parallel 1
```

If you did not use the /PARALLEL qualifier, VAX C predefines CC\$parallel as if the following line appeared at the top of the compilation unit:

```
#define CC$parallel 0
```

You can use this macro to take advantage conditionally of either parallel-processing features or features that would disable the decomposition of a loop. The following example shows how you can place a function call inside a loop on the condition that parallel processing is not taking place:

```
for (i = 0; i < 1000; ++i)
{
    a[i] = b[i] * c[i];
    #if !CC$parallel
        /* Avoid doing optional function call that would
         * inhibit loop decomposition.
         */
        printf("%6.2f\n", a[i]);
    #endif
}
```

11.1.3 The `__DATE__` Macro

The `__DATE__` macro evaluates to a string specifying the date on which the compilation started. The string presents the date in the following format:

Mmm-dd-yyyy

The first d is a space if dd is less than 10.

The following is an example of the `__DATE__` macro:

```
printf("%s", __DATE__);
```

11.1.4 The `__FILE__` Macro

The `__FILE__` macro evaluates to a string specifying the file specification of the current source file. The following is an example of the `__FILE__` macro:

```
printf("file %s" __FILE__);
```

11.1.5 The `__LINE__` Macro

The `__LINE__` macro evaluates to an integer specifying the number of the line in the source file containing the macro reference. The following is an example of the `__LINE__` macro:

```
printf("At line %d in file %s", __LINE__, __FILE__);
```

11.1.6 The `__TIME__` Macro

The `__TIME__` macro evaluates to a string specifying when the compilation started. The string presents the time in the following format:

hh:mm:ss

The following is an example of the `__TIME__` macro:

```
printf("%s", __TIME__);
```

11.1.7 vax, vms, vaxc, and vax11c (System-Identification Macros)

VAX C automatically defines macros that can be used to identify the system on which the program is running. These macros can assist in writing code that executes conditionally, depending on whether the program is running on a DIGITAL system or some other system. These symbols are defined as if the following text fragment were included by the compiler before every compilation source group:

```
#define vax          1
#define VAX         1
#define vms          1
#define VMS         1
#define vaxc        1
#define VAXC        1
#define vax11c     1
#define VAX11C     1
```

You can use these definitions to separate portable and nonportable code in any of your VAX C programs.

You can use the symbols to conditionally compile VAX C programs used on more than one operating system to take advantage of system-specific features. See Section 10.3 for more information about using the preprocessor conditional-compilation directives.

Consider the following example:

```
#if      VAXC
#include  rms          /* Include RMS definitions */
#endif
```

11.2 Built-In Functions

The following sections describe the built-in functions that allow you to directly access the VAX hardware and machine instructions to perform operations that are cumbersome, slow, or impossible in pure C.

These functions are very efficient because they are built into the VAX C compiler. This means that a call to one of these functions does not result in a reference to a function in the VAX C Run-Time Library (RTL) or to a function in your program. Instead, the compiler generates the machine instructions necessary to carry out the function directly at the call site. Because most of these built-in functions closely correspond to single VAX machine instructions, the result is small, fast code.

Some of these built-in functions (such as those that operate on strings or bits) are of general interest. Others (such as the functions dealing with process context) are of interest if you are writing device drivers or other privileged software. Some of the functions discussed in the following sections are privileged and unavailable to user mode programs.

You must place the following pragma in your source file before using one or more built-in functions:

```
#pragma builtins
```

Some of the built-in functions have optional arguments or allow a particular argument to have one of many different types. To describe the different legal combinations of arguments, the description of each built-in function may list several different prototypes for the function. As long as a call to a built-in function matches one of the prototypes listed, the call is legal. Furthermore, any legal call to a built-in function acts as if the corresponding prototype were in scope. Thus, the compiler performs the argument checking and argument conversions specified by that prototype.

The majority of the built-in functions are named after the VAX instruction that they generate. The built-in functions provide direct and unencumbered access to those VAX instructions. Any inherent limitations to those instructions are limitations to the built-in functions as well. For instance, the MOVC3 instruction and the `_MOVC3` built-in function can move at most 65,535 characters.

For more information on these built-ins, see the documentation on the corresponding machine instruction in the *VAX MACRO and Instruction Set Reference Manual*. In particular, see that book for the structure of queue entries manipulated by the built-in queue functions.

11.2.1 Add Aligned Word Interlocked (`_ADAWI`)

The `_ADAWI` function adds its source operand to the destination. This function is interlocked against similar operations by other processors or devices in the system.

The `_ADAWI` function has the following formats:

```
int _ADAWI(short src, short *dest);
int _ADAWI(short src, unsigned short *dest);
```

src

Is the value to be added to the destination.

dest

Is a pointer to the destination. The destination must be aligned on a word boundary. (One way to achieve alignment is to use **_align**.)

There are three possible return values, as follows:

- -1, if the sum when considered to be a signed number is negative
- 0, if the sum is zero
- 1, if the sum is positive

11.2.2 Branch on Bit Clear-Clear Interlocked (**_BBCCI**)

The **_BBCCI** function performs the following functions in interlocked fashion:

- Returns the complement of the bit specified by the two arguments
- Clears the bit specified by the two arguments

The **_BBCCI** function has the following format:

```
int _BBCCI(int position, void *address);
```

position

Is the position of the bit within the field.

address

Is the base address of the field.

The return value is 0 or 1, which is the complement of the value of the specified bit before being cleared.

11.2.3 Branch on Bit Set-Set Interlocked (**_BBSSI**)

The **_BBSSI** function performs the following functions in interlocked fashion:

- Returns the status of the bit specified by the two arguments
- Sets the bit specified by the two arguments

The **_BBSSI** function has the following format:

```
int _BBSSI(int position, void *address);
```

position

Is the position of the bit within the field.

address

Is the base address of the field.

The return value is 0 or 1, which is the value of the specified bit before being set.

11.2.4 Find First Clear Bit (`_FFC`)

The `_FFC` function finds the position of the first clear bit in a field. The bits are tested for clear status starting at bit 0 and extending to the highest bit in the field.

The `_FFC` function has the following format:

```
int _FFC(int start, char size, const void *base, int *position);
```

start

Is the start position of the field.

size

Is the size of the field, in bits. The size must be a value from 0 to 32 bits.

base

Is the address of the field.

position

Is the address of an integer to receive the position of the clear bit. If no bit is clear, the integer is set to the position of the first bit to the left of the last bit tested.

There are two possible return values, as follows:

- 0, if all bits in the field are set
- 1, if a bit with value 0 is found

11.2.5 Find First Set Bit (`_FFS`)

The `_FFS` function finds the position of the first set bit in a field. The bits are tested for set status starting at bit 0 and extending to the highest bit in the field.

The `_FFS` function has the following format:

```
int _FFS(int start, char size, const void *base, int *position);
```

start

Is the start position of the field.

size

Is the size of the field, in bits. The size must be a value from 0 to 32 bits.

base

Is the address of the field.

position

Is the address of an `int` to receive the position of the set bit. If no bit is set, the integer is set to the position of the first bit to the left of the last bit tested.

There are two possible return values, as follows:

- 0, if all bits in the field are clear
- 1, if a bit with value 1 is found

11.2.6 Halt (`_HALT`)

The `_HALT` function halts the processor when executed by a process running in kernel mode. This is a privileged function.

The `_HALT` function has the following format:

```
void _HALT(void);
```

11.2.7 Insert Entry into Queue at Head Interlocked (`_INSQHI`)

The `_INSQHI` function inserts an entry into the front of a queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The `_INSQHI` function has the following format:

```
int _INSQHI(void *new_entry, void *head);
```

new_entry

Is a pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary. (One way to achieve alignment is to use `_align`.)

head

Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use `_align`.)

There are three possible return values, as follows:

- 0, if the entry was inserted, but it was not the only entry in the list
- 1, if the entry was not inserted because the secondary interlock failed
- 2, if the entry was inserted and it was the only entry in the list

11.2.8 Insert Entry into Queue at Tail Interlocked (`_INSQTI`)

The `_INSQTI` function inserts an entry at the end of a queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The `_INSQTI` function has the following format:

```
int _INSQTI(void *new_entry, void *head);
```

new_entry

Is a pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary. (One way to achieve alignment is to use `_align`.)

head

Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use `_align`.)

There are three possible return values, as follows:

- 0, if the entry was inserted, but it was not the only entry in the list
- 1, if the entry was not inserted because the secondary interlock failed

- 2, if the entry was inserted and it was the only entry in the list

11.2.9 Insert Entry in Queue (`_INSQUE`)

The `_INSQUE` function inserts a new entry into a queue following an existing entry.

The `_INSQUE` function has the following format:

```
int _INSQUE(void *new_entry, void *predecessor);
```

new_entry

Is a pointer to the new entry to be inserted.

predecessor

Is a pointer to an existing entry in the queue.

There are two possible return values, as follows:

- 0, if the entry was the only entry in the queue
- 1, if the entry was not the only entry in the queue

11.2.10 Load Process Context (`_LDPCTX`)

The `_LDPCTX` function restores the register and memory-management context. This is a privileged function.

The `_LDPCTX` function has the following format:

```
void _LDPCTX(void);
```

11.2.11 Locate Character (`_LOCC`)

The `_LOCC` function locates the first character in a string matching the target character.

The `_LOCC` function has the following formats:

```
int _LOCC(char target, unsigned short length,  
          const char *string);
```

```
int _LOCC(char target, unsigned short length,  
          const char *string, char **position);
```

target

Is the character being searched.

length

Is the length of the searched string. The length must be a value from 0 to 65,535.

string

Is a pointer to the searched string.

position

Is a pointer to a pointer to a character. If the searched character is found, the pointer pointed to by position is updated to point to the character found. If the character is not found, the pointer pointed to by position is set to the address one byte beyond the string. This is an optional argument.

If the target character is found, the return value is the number of bytes remaining in the string; otherwise, the return value is 0.

11.2.12 Move from Processor Register (**_MFPR**)

The **_MFPR** function returns the contents of a processor register. This is a privileged function.

The **_MFPR** function has the following formats:

```
void _MFPR(int register_num, int *destination);  
void _MFPR(int register_num, unsigned int *destination);
```

register_num

Is the number of the privileged register to be read.

destination

Is a pointer to the location receiving the value from the register. This location may be a **signed** or **unsigned int**.

11.2.13 Move Character 3 Operand (**_MOVC3**)

The **_MOVC3** function copies a block of memory. It is the preferred way to copy a block of memory to a new location.

The **_MOVC3** function has the following formats:

```
void _MOVC3(unsigned short length, const char *src, char *dest);
```



```
void _MOVC3(unsigned short length, const char *src, char *dest,  
            char **endsrc);
```

```
void _MOVC3(unsigned short length, const char *src, char *dest,  
            char **endsrc, char **enddest);
```

length

Is the length of the source string, in bytes. The length must be a value from 0 to 65,535.

src

Is a pointer to the source string.

dest

Is a pointer to the destination memory.

endsrc

Is a pointer to a pointer. The `_MOVC3` function sets the pointer that is pointed to by `endsrc` pointing to the address of the byte beyond the source string. It is optional if the `enddest` argument is not given.

enddest

Is a pointer to a pointer. The `_MOVC3` function sets the pointer pointed to by `endsrc` to the address of the byte beyond the destination string. This is an optional argument.

11.2.14 Move Character 5 Operand (`_MOVC5`)

The `_MOVC5` function allows the source string specified by the pointer and length pair to be moved to the destination string specified by the other pointer and length pair. If the source string is smaller than the destination string, the destination string is padded with the specified character.

The `_MOVC5` function has the following formats:

```
void _MOVC5(unsigned short srclen, const char *src, char fill,  
            unsigned short destlen, char *dest);
```

```
void _MOVC5(unsigned short srclen, const char *src, char fill,  
            unsigned short destlen, char *dest,  
            unsigned short *unmoved_src);
```

```
void _MOVC5(unsigned short srclen, const char *src, char fill,  
            unsigned short destlen, char *dest,  
            unsigned short *unmoved_src, char **endsrc);
```

```
void _MOVC5(unsigned short srclen, const char *src, char fill,
            unsigned short destlen, char *dest,
            unsigned short *unmoved_src, char **endsrc,
            char **enddest);
```

srclen

Is the length of the source string, in bytes. The length must be a value from 0 to 65,535.

src

Is a pointer to the source string.

fill

Is the fill character to be used if the source string is smaller than the destination string.

destlen

Is the length of the destination string, in bytes. The length must be a value from 0 to 65,535.

dest

Is a pointer to the destination string.

unmoved_src

Is a pointer to a short integer that the `_MOVC5` function sets to the number of unmoved bytes remaining in the source string.

endsrc

Is a pointer to a pointer. The `_MOVC5` function sets the pointer pointed to by `endsrc` pointing to the address of the byte beyond the source string. It is optional if the `enddest` argument is not given.

enddest

Is a pointer to a pointer. The `_MOVC5` function sets the pointer pointed to by `endsrc` to the address of the byte beyond the destination string. This is an optional argument.

11.2.15 Move from Processor Status Longword (`_MOVPSL`)

The `_MOVPSL` function stores the value of the Processor Status Longword (PSL).

The `_MOVPSL` function has the following formats:

```
void _MOVPSL(int *psl);  
void _MOVPSL(unsigned int *psl);
```

psl

Is the address of the location for storing the value of the Processor Status Longword.

11.2.16 Move to Processor Register (`_MTPR`)

The `_MTPR` function loads a value into one of the special processor registers. It is a privileged function.

The `_MTPR` function has the following format:

```
int _MTPR(int src, int register_num);
```

src

Is the value to store into the processor register.

register_num

Is the number of a privileged register to be updated.

The return value is the V condition flag from the Processor Status Longword (PSL).

11.2.17 Probe Read Accessibility (`_PROBER`)

The `_PROBER` function checks to see if you can read the first and last byte of the given address and length pair.

The `_PROBER` function has the following format:

```
int _PROBER(char mode, unsigned short length, const void *address);
```

mode

Is the processor mode used for checking the access.

length

Is the length of the memory segment, in bytes. The length must be a value from 0 to 65,535.

address

Is the pointer to the memory segment to be tested for read access.

There are two possible return values, as follows:

- 0, if both bytes are not accessible
- 1, if both bytes are accessible

11.2.18 Probe Write Accessibility (`_PROBEW`)

The `_PROBEW` function checks the write accessibility of the first and last byte of the given address and length pair.

The `_PROBEW` function has the following format:

```
int _PROBEW(char mode, unsigned short length, const void *address);
```

mode

Is the processor mode used for checking the access.

length

Is the length of the memory segment, in bytes. The length must be a value from 0 to 65,535.

address

Is the pointer to the memory segment to be tested for write access.

There are two possible return values, as follows:

- 0, if both bytes are not accessible
- 1, if both bytes are accessible

11.2.19 Read General-Purpose Register (`_READ_GPR`)

The `_READ_GPR` function returns the value of a general-purpose register.

The `_READ_GPR` function has the following format:

```
int _READ_GPR(int register_num);
```

register_num

Is an integer constant expression giving the number of the general-purpose register to be read.

The return value is the value of the general-purpose register.

11.2.20 Remove Entry from Queue at Head Interlocked (`_REMQHI`)

The `_REMQHI` function removes the first entry from the queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The `_REMQHI` function has the following format:

```
int _REMQHI(void *head, void **removed_entry);
```

head

Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use `_align`.)

removed_entry

Is a pointer to a pointer that `_REMQHI` sets to point to the removed entry.

There are four possible return values, as follows:

- 0, if the entry was removed and the queue has remaining entries
- 1, if the entry could not be removed because the secondary interlock failed
- 2, if the entry was removed and the queue is now empty
- 3, if the queue was empty

11.2.21 Remove Entry from Queue at Tail Interlocked (`_REMQTI`)

The `_REMQTI` function removes the last entry from the queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The `_REMQTI` function has the following format:

```
int _REMQTI(void *head, void **removed_entry);
```

head

Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use `_align`.)

removed_entry

Is a pointer to a pointer that `_REMQTI` sets to point to the removed entry.

There are four possible return values, as follows:

- 0, if the entry was removed and the queue has remaining entries

- 1, if the entry could not be removed because the secondary interlock failed
- 2, if the entry was removed and the queue is now empty
- 3, if the queue was empty

11.2.22 Remove Entry from Queue (`_REMQUE`)

The `_REMQUE` function removes an entry from a queue.

The `_REMQUE` function has the following format:

```
int _REMQUE(void *entry, void **removed_entry);
```

entry

Is a pointer to the queue entry to be removed.

removed_entry

Is a pointer to a pointer that `_REMQUE` sets to the address of the entry removed from the queue.

There are three possible return values, as follows:

- 0, if the entry was removed and the queue has remaining entries
- 1, if the entry was removed and the queue is now empty
- 2, if the queue was empty

11.2.23 Scan Characters (`_SCANC`)

The `_SCANC` function locates the first character in a string with the desired attributes. The attributes are specified through a table and a mask.

The `_SCANC` function has the following formats:

```
int _SCANC(unsigned short length, const char *string,  
           const char *table, char mask);
```

```
int _SCANC(unsigned short length, const char *string,  
           const char *table, char mask, char **match);
```

length

Is the length of the string to scan, in bytes. The length must be a value from 0 to 65,535.

string

Is a pointer to the string to scan.

table

Is a pointer to the table.

mask

Is the mask.

match

Is a pointer to a pointer that the `_SCANC` function sets to the address of the byte that matched. (If no match occurs, it is set to the address of the byte following the string.) This is an optional argument.

The return value is the number of bytes remaining in the string if a match was found; otherwise, the return value is 0.

11.2.24 Simple Read (`_SIMPLE_READ`)

The `_SIMPLE_READ` function reads I/O registers or shared memory. It causes a `MOVB`, `MOVW`, or `MOVL` instruction to be generated that cannot be moved or modified during optimization.

The `_SIMPLE_READ` function has the following formats:

```
char _SIMPLE_READ(const char *source);
short _SIMPLE_READ(const short *source);
int _SIMPLE_READ(const int *source);
long _SIMPLE_READ(const long *source);
```

source

Is a pointer to the source to be read. The object being pointed to must be a signed integer. The type of the object pointed to determines the type of the function result.

The return value is the value of the specified source.

11.2.25 Simple Write (`_SIMPLE_WRITE`)

The `_SIMPLE_WRITE` function writes to I/O registers or shared memory. It causes a `MOVB`, `MOVW`, or `MOVL` instruction to be generated that cannot be moved or modified during optimization.

The `_SIMPLE_WRITE` function has the following formats:

```
void _SIMPLE_WRITE(char value, char *dest);
void _SIMPLE_WRITE(short value, short *dest);
void _SIMPLE_WRITE(int value, int *dest);
void _SIMPLE_WRITE(long value, long *dest);
```

value

Is the value to be stored. The type of the destination argument determines the type of this argument.

dest

Is a pointer to the destination. The type of the object pointed to by `dest` must be a signed integer type. The type of this object determines the type of the first argument to this function.

11.2.26 Skip Character (`_SKPC`)

The `_SKPC` function locates the first character in a string that does not match the target character.

The `_SKPC` function has the following formats:

```
int _SKPC(char target, unsigned short length, const char *string);
int _SKPC(char target, unsigned short length, const char *string,
          char **position);
```

target

Is the target character.

length

Is the length of the string, in bytes. The length must be a value from 0 to 65,535.

string

Is a pointer to the string to scan.

position

Is a pointer to a pointer. The `_SKPC` function sets the pointer pointed to by `position` to the address of the nonmatching character. (If all the characters match, it is set to the address of the first byte beyond the string.) This is an optional argument.

The return value is the number of bytes remaining in the string if an unequal byte was located; otherwise, the return value is 0.

11.2.27 Span Characters (`_SPANC`)

The `_SPANC` function locates the first character in a string without certain attributes. The attributes are specified through a table and a mask.

The `_SPANC` function has the following formats:

```
int _SPANC(unsigned short length, const char *string,  
           const char *table, char mask);
```

```
int _SPANC(unsigned short length, const char *string,  
           const char *table, char mask, char **position);
```

length

Is the length of the string, in bytes. The length must be a value from 0 to 65,535.

string

Is a pointer. It points to the string to be scanned.

table

Is a pointer to the table.

mask

Is the mask.

position

Is a pointer to a pointer. The `_SPANC` function sets the pointer pointed to by `position` to the address of the byte that does not match the attributes. (If all the characters in the string match, this pointer is set to the address of the first byte beyond the string.) This is an optional argument.

The return value is the number of bytes remaining in the string if a match was found; otherwise, the return value is 0.

11.2.28 Save Process Context (`_SVPCTX`)

The `_SVPCTX` function saves the context of a process. The general-purpose registers are saved in the process control block, which is later used to resume a process. This function is privileged.

The `_SVPCTX` function has the following format:

```
void _SVPCTX(void);
```

11.2.29 Write General-Purpose Register (`_WRITE_GPR`)

The `_WRITE_GPR` function loads a value into a specified general-purpose register.

The `_WRITE_GPR` function has the following format:

```
void _WRITE_GPR(int value, int register_num);
```

value

Is the value to load into the register.

register_num

Is an integer constant expression giving the number of the general-purpose register to be loaded. The register number must be a value from 0 to 15.

Using VAX C Features on VMS Systems



Using VAX Record Management Services

VAX C provides a set of run-time library functions and macros to perform I/O. Some of these functions perform in the same manner as I/O functions found on C implementations running on UNIX systems. Other VAX C functions take full advantage of the functionality of the VMS file-handling system. You can also access the VMS file-handling system from your VAX C program without using the VAX C Run-Time Library (RTL) functions. In any case, the system that ultimately accesses files on VMS systems is VAX Record Management Services (RMS).

This chapter introduces you to the following RMS topics:

- RMS file organization (Section 12.1)
- Sequential file organization (Section 12.1.1)
- Relative file organization (Section 12.1.2)
- Indexed file organization (Section 12.1.3)
- Record access modes (Section 12.2)
- RMS record formats (Section 12.3)
- RMS functions (Section 12.4)
- Writing VAX C programs using RMS (Section 12.5)
- RMS example program (Section 12.6)

The file-handling capabilities of VAX C fall into two distinct categories:

- The VAX C RTL functions which, with little or no modification, are portable to other C implementations
- The RMS functions, which are not portable to other C implementations but do provide more methods of file organization and more record access modes

This chapter briefly reviews the basic concepts and facilities of VAX RMS and shows examples of their application in VAX C programming. Because this is an overview, the chapter does not explain all RMS concepts and features. For language-independent information about RMS, see the following manuals in the VMS document set:

- *Guide to VMS File Applications*
This guide contains a general description of the record management services of the VMS operating system, and the file creation and run-time options available.
- *VMS Record Management Services Manual*
This manual describes the user interface to RMS. It includes introductory information on RMS programming and detailed definitions of all RMS control block structures and macro instructions.

12.1 RMS File Organization

VAX RMS supports three types of file organization:

- Sequential
- Relative
- Indexed

The following sections describe these types of file organization.

The organization of a file determines how a file is stored on the media and, consequently, the possible operations on records. You specify the file's organization when you create the file; it cannot be changed.

However, you can use the File Definition Language Editor (FDL) and the CONVERT or CONVERT/RECLAIM utilities to define the characteristics of a new file, and then fill the new file with the contents of the old file of a different format. For more information, see the *VMS Utility Routines Manual*.

12.1.1 Sequential File Organization

Sequential files have consecutive records. There are no empty records separating records that contain data. This organization allows the following operations on the file:

- Positioning the file at a particular record, generally by sequentially moving from one record to the next.

Direct access is also possible, either by key (relative record number) or by the record file address (RFA). However, although allowed for any file organization, access by RFA is limited to files on disk devices, and access by key is limited to disk files that also have fixed-length records. These access modes are unusual because most application programs do not keep track of record positions in sequential files.

- Reading data from any record.
- Writing data by adding records at the end of the file.

Sequential organization is the only kind permitted for magnetic tape files and other nondisk devices.

12.1.2 Relative File Organization

Relative files have records that occupy numbered, fixed-length cells. The records themselves need not have the same length. Cells can be empty or can contain records so the following operations are permitted:

- Positioning the file at a particular record, usually by direct access.
In direct access, RMS uses the relative record number—the number of a cell—as a key to locate the cell and its record; there is no need to reference other cells. RMS can also access the records sequentially, ignoring empty cells, or RMS can access the file directly with the record file address (RFA); RMS returns the RFA in a parameter block whenever it writes a record, and you can access and use the RFA to locate the appropriate record. You can access any file organization with the RFA.
- Reading a record from any cell.
- Deleting a record from any cell.
- Writing a record into any cell.

Relative file organization is possible only on disk devices.

12.1.3 Indexed File Organization

Indexed files have records that contain, in addition to data and carriage-control information, one or more keys. Keys can be character strings, packed decimal numbers, and 16-bit, 32-bit, or 64-bit signed or unsigned integers. Every record has at least one key, the primary key, whose value in each record cannot be changed. Optionally, each record can have one or more alternate keys, whose key values can be changed.

Unlike relative record numbers used in relative files, key values in indexed files are not necessarily unique. When you create a file, you can specify that a particular key have the same value in different records (these keys are called duplicate keys). Keys are defined for the entire file, in terms of their position within a record and their length.

In addition to maintaining its records, RMS builds and maintains indexes for each of the defined keys. As records are written to the file, their key values are inserted in order of ascending value in the appropriate indexes. This organization allows the following operations:

- Positioning the file at a particular record, by direct access. In direct access reads, you use either a primary or alternate key, plus a specified key value, to locate the record. In direct access writes (given a record that contains key values in the predefined positions), RMS automatically adds the record to the file and adds the primary and alternate key values to the appropriate indexes. Records can also be accessed sequentially, where the sequence is defined by the index for a specified key. Finally, records can be accessed directly by RFA; RMS returns the RFA in a parameter block whenever it writes a record, and you can access and use the RFA to locate the appropriate record. You can access any file organization with the RFA.
- Reading any record, including sequential reads controlled by a key's index.
- Deleting any record.
- Updating an alternate key's value, if the key's definition permits its value to change.
- Writing records selectively, based on the value of a key and, when allowed in the key's definition, based on duplicate values. If duplicate values are permitted, you can write records containing key values that are present in the key's index. If duplicate values are not permitted, such write operations are rejected.

Indexed organization is possible only on disk devices.

12.2 Record Access Modes

The record access modes are sequential, direct by key, and direct by record file address. Again, the direct access modes are possible only with files that reside on disks.

Unlike a file's organization, the record access mode is not a permanent attribute of the file. During the processing of a file, you can switch from one access mode to any other permitted for that file organization. For example, indexed files are often processed by locating a record directly by key, and then using that key's index to sequentially read all the indexed records in ascending order of their key values; this method is sometimes called the indexed-sequential access method (ISAM).

12.3 RMS Record Formats

Records in RMS files can have the following formats:

- Fixed-length format, where the length of every record is defined at the time of the file's creation. This format is permitted with any file organization.
- Variable-length format, where the maximum length of every record is defined at the time of the file's creation. This format is permitted with any file organization.
- Variable-length format with a fixed-length control area (VFC), where every record is prefixed by a fixed-length field. This format is permitted only with sequential and relative files.
- Stream format, where records are delimited by special characters called *terminators*. Terminators are part of the record they delimit. The three types of stream formatting are as follows:
 - Stream variation, where records can be delimited with any special character.
 - Stream_cr, where records are delimited with the carriage-return character.
 - Stream_lf, where records are delimited with the line-feed character. This format variation is the default format when you create files using the Standard I/O functions.

12.4 RMS Functions

RMS provides a number of functions that create and manipulate files. These functions use RMS data structures to define the characteristics of a file and its records. The data structures thus are used as indirect arguments to the function call.

The RMS data structures are grouped into four main categories, as follows:

- **File Access Block (FAB)**
Defines the file's characteristics, such as file organization and record format.
- **Record Access Block (RAB)**
Defines the way in which records are processed, such as the record access mode.
- **Extended Attribute Block (XAB)**
Various kinds of extended attribute blocks contain additional file characteristics, such as the definition of keys in an indexed file. Extended attribute blocks are optional.
- **Name Block (NAM)**
Defines all or part of a file specification to be used when an incomplete file specification is given in an OPEN or CREATE operation. Name blocks are optional.

RMS uses these data structures to perform file and record operations. Table 12–1 lists some of the common functions.

Table 12–1: Common RMS Run-Time Processing Functions

Category	Function	Description
File Processing	sys\$create	Creates and opens a new file of any organization.
	sys\$open	Opens an existing file and initiates file processing.
	sys\$close	Terminates file processing and closes the file.
Record Processing	sys\$erase	Deletes a file.
	sys\$connect	Associates a file access block with a record access block to establish a record access stream; a call to this function is required before any other record-processing function can be used.
	sys\$get	Retrieves a record from a file.
	sys\$put	Writes a new record to a file.
	sys\$update	Rewrites an existing record to a file.
	sys\$delete	Deletes a record from a file.

(continued on next page)

Table 12–1 (Cont.): Common RMS Run-Time Processing Functions

Category	Function	Description
	sys\$rewind	Positions the record pointer to the first record in the file.
	sys\$disconnect	Disconnects a record access stream.

All RMS functions are directly accessible from VAX C programs. The syntax for any RMS function has the following form:

```
int      sys$name(pointer)
struct  rms_structure *pointer;
```

In this syntax, name corresponds to the name of the RMS function (such as OPEN or CREATE); rms_structure corresponds to the name of the structure being used by the function.

The file-processing functions require a pointer to a file access block as an argument; the record-processing functions require a pointer to a record access block as an argument. Since sys\$create is a file-processing function, its syntax is as follows:

```
int      sys$create(fab)
struct  struct FAB *fab;
```

These syntax descriptions do not show all the options available when you invoke an RMS function. For a complete description of the RMS calling sequence, see the *VMS Record Management Services Manual*.

Finally, all the RMS functions return an integer status value. The format of RMS status values follows the standard format described in Chapter 13. Since they return a 32-bit integer, you do not need to declare the type of an RMS function return before you use it.

12.5 Writing VAX C Programs Using RMS

VAX C supplies a number of **#include** modules that describe the RMS data structures and status codes. Table 12–2 lists these modules.

Table 12–2: VAX C RMS #include Modules

Module Name	Structure Tag(s)	Description
fab	FAB	Defines the file access block structure.
rab	RAB	Defines the record access block structure.
nam	NAM	Defines the name block structure.
xab	XAB	Defines all the extended attribute block structures.
rmsdef	-	Defines the completion status codes that RMS returns after every file- or record-processing operation.
rms	all tags	Includes all the previous modules.

Most VAX C programmers include the **rms** module, which includes all the other modules.

These **#include** modules define all the data structures as structure tag names. However, they perform no allocation or initialization of the structures; these modules describe only a template for the structures. To use the structures, you must create storage for them and initialize all the structure members as required by RMS. Note that these include files are part of VAX C. RMS is part of VMS and may contain other include files not described here.

To assist in the initialization process, VAX C provides initialized RMS data structure prototypes. You can copy these **readonly** prototypes to your uninitialized structure definitions with a structure assignment. You can choose to take the default values for each of the structure members (as initialized by the prototypes), or you can tailor the contents of the structures to fit your requirements. In either case, you must use the templates to allocate storage for the structure and to define the members of the structure.

The initialized prototypes supply the RMS default values for each member in the structure; they specify none of the optional parameters. To determine what default values are supplied by the prototypes, see the *VMS Record Management Services Manual*.

Table 12–3 lists the prototype data structures and the structures that they initialize.

Table 12–3: RMS Prototype Data Structures

Prototype	Structure Tag	Initialize Structure
cc\$rms_fab	FAB	File access block
cc\$rms_rab	RAB	Record access block
cc\$rms_nam	NAM	Name block
cc\$rms_xaball	XABALL	Allocation extended attribute block
cc\$rms_xabdat	XABDAT	Date and time extended attribute block
cc\$rms_xabfhc	XABFHC	File header characteristics extended attribute block
cc\$rms_xabkey	XABKEY	Indexed file key extended attribute block
cc\$rms_xabpro	XABPRO	Protection extended attribute block
cc\$rms_xabrtdt	XABRDT	Revision date and time extended attribute block
cc\$rms_xabsum	XABSUM	Summary extended attribute block

The declarations of these structures are contained in the appropriate **#include** module.

The names of the structure members conform to the following RMS naming convention:

typ\$s_fld

The identifier *typ* is the abbreviation for the structure, the letter *s* is the size of the member (such as *l* for longword or *b* for byte), and the identifier *fld* is the member name, such as *sts* for the completion status code. The dollar sign (\$) is a character used in VMS system logical names. See the *VMS Record Management Services Manual* for a description of the members in each structure.

12.5.1 Initializing File Access Blocks

The file access block defines the attributes of the file. To initialize a file access block, assign the values in the initialized data structure `cc$rms_fab` to the address of the file access block defined in your program. Consider the following example:

```

/* This example shows how to initialize a file access block. */
#include rms                /* Declare all RMS data structs */
struct FAB  fblock;        /* Define a file access block */
main()
{
    fblock = cc$rms_fab;    /* Initialize the structure */
    .
    .
}

```

Any of these RMS structures may be dynamically allocated. For example, another way to allocate a file access block is as follows:

```

/* This program shows how to dynamically allocate RMS structures. */
#include rms                /* Declare all RMS data structs */
main()
{
    /* Allocate dynamic storage */
    struct FAB  *fptr = malloc(sizeof (struct FAB));
    *fptr = cc$rms_fab;    /* Initialize the structure */
    .
    .
}

```

Frequently, you will want to change the default values supplied by the prototype. If so, you must reinitialize the members of the structure individually. You initialize a member by giving the offset of the member and assigning a value to it. Consider the following example:

```
fblock.fab$l_xab = &primary_key;
```

This statement assigns the address of the extended attribute block named `primary_key` to the `fab$l_xab` member of the file access block named `fblock`.

12.5.2 Initializing Record Access Blocks

The record access block specifies how records are processed. You initialize a record access block in the same manner as you initialize a file access block. For example:

```

/* This example shows how to initialize a file access block. */
#include rms
struct FAB  fblock;
struct RAB  rblock;        /* Define a record access block */

```

```

main()
{
    fblock = cc$rms_fab;          /* Initialize the structure */
    rblock = cc$rms_rab;

                                /* Initialize the FAB member */
    rblock.rab$l_fab = &fblock;
    .
    .
}

```

12.5.3 Initializing Extended Attribute Blocks

There is only one extended attribute block structure (XAB), but there are seven ways to initialize it. The extended attribute blocks define additional file attributes that are not defined elsewhere. For example, the key extended attribute block is used to define the keys of an indexed file.

All extended attribute blocks are “chained” off a file access block in the following manner:

1. In a file access block, you initialize the `fab$l_xab` field with the address of the first extended attribute block.
2. You designate the next extended attribute block in the chain in the `xab$l_nxt` field of any subsequent extended attribute blocks. You chain each subsequent extended attribute block in order by the key of reference (first the primary key, then the first alternate key, then the second alternate key, and so forth).
3. You initialize the `xab$l_nxt` member of the last extended attribute block in the chain with the value 0 (the default), to indicate the end of the chain.

You go through the same steps to declare extended attribute blocks as you would to declare the other RMS data structures:

1. You define the structures with **#include** modules.
2. You assign a specific prototype to the structure in your program.
3. You initialize the members of the structure with the desired values.

In the following example, two extended attribute block structures are declared. They are initialized as key extended attribute blocks with the `cc$rms_xabkey` prototype. The `xab$l_nxt` member of the primary key is initialized with the address of the `alternate_key` extended attribute block.


```

/* This example shows how to initialize the extended      *
 * attribute block.                                       */

#include rms
struct XABKEY primary_key, alternate_key;

main()
{
    primary_key          = cc$rms_xabkey;
    alternate_key        = cc$rms_xabkey;
    primary_key.xab$l_nxt = &alternate_key;
    .
    .
}

```

12.5.4 Initializing Name Blocks

The name block contains default file name values, such as the directory or device specification, file name, or file type. If you do not specify one of the parts of the file specification when you open the file, RMS uses the values in the name block to complete the file specification and places the complete file specification in an array.

You create and initialize name blocks in the same manner used to initialize the other RMS data structures. Consider the following example:

```

/* This example shows how to initialize a name block.      */
#include rms
struct NAM nam;
struct FAB fab;

main()
{
    fab = cc$rms_fab;
    nam = cc$rms_nam;

                                /* Define an array for the      *
                                *   expanded file specification */
    char expanded_name[NAM$C_MAXRSS];

                                /* Initialize the appropriate   *
                                *   members                       */

    fab.fab$l_nam = &nam;
    nam.nam$l_esa = &expanded_name;
    nam.nam$b_ess = sizeof expanded_name;
    .
    .
}

```

12.6 RMS Example Program

The example program in this section uses RMS functions to maintain a simple employee file. The file is an indexed file with two keys: social security number and last name. The fields in the record are character strings defined in a structure with the tag record.

The records have the carriage-return attribute. Individual fields in each record are padded with blanks for two reasons. First, those fields that are key fields must be padded in some way; RMS does not understand VAX C strings with the trailing NUL character. Second, the choice of blank padding as opposed to NUL padding allows the file to be printed or typed without conversion.

The program does not perform range or bounds checking. Only the error checking that shows the mapping of VAX C to RMS is performed. Any other errors are considered fatal.

The program is divided into the following sections:

- External data declarations and definitions
- Main program section
- Function to initialize the RMS data structures
- Internal functions to open the file, display HELP information, pad the records, and process fatal errors
- Utility functions
 - ADD
 - DELETE
 - TYPE
 - PRINT
 - UPDATE

To run this program, perform the following steps:

1. Create a source file. The name of the source file in this example is RMSEXP.C. For more information about creating source files, see Chapter 1.
2. Compile the source file with the following command:

```
$ CC RMSEXP 
```

For more information about the compiling process, see Chapter 1.

3. Link the program with the following command:

```
$ LINK RMSEXP, SYS$LIBRARY:VAXCTRL/LIB RETURN
```

For more information about the linking process, see Chapter 1.

4. Because the program expects command line arguments, it must be defined as a foreign command. You can do this with the following command line:

```
$ RMSEXP ::= $device:[directory]RMSEXP RETURN
```

The identifier `device` is the logical or physical name of the device containing your directory; the identifier `directory` is the name of your directory. The device name must be preceded by the dollar sign (\$) to be recognized as a foreign command by the DCL interpreter.

For more information about foreign commands, see Chapter 1.

5. Run the program using the following foreign command:

```
$ RMSEXP filename RETURN
```

The complete listing (by section) of the example program follows. Notes on each section are keyed to the numbers in the listing.

Example 12–1 shows the external data declarations and definitions.

Example 12–1: External Data Declarations and Definitions

```
/* This segment of RMSEXP.C contains external data      *  
 * definitions.                                         */  
  
① #include rms  
   #include stdio  
   #include ssdef  
  
② #define  DEFAULT_FILE_NAME      ".dat"  
  
   #define  RECORD_SIZE           (sizeof record)  
   #define  SIZE_SSN              15  
   #define  SIZE_LNAME            25  
   #define  SIZE_FNAME            25  
   #define  SIZE_COMMENTS        15  
   #define  KEY_SIZE              \  
   (SIZE_SSN > SIZE_LNAME ? SIZE_SSN: SIZE_LNAME)  
  
③ struct  FAB fab;  
   struct  RAB rab;  
   struct  XABKEY primary_key, alternate_key;
```

(continued on next page)

Example 12–1 (Cont.): External Data Declarations and Definitions

```
④ struct
  {
    char    ssn[SIZE_SSN], last_name[SIZE_LNAME];
    char    first_name[SIZE_FNAME],
           comments[SIZE_COMMENTS];
  } record;

⑤ char  response[BUFSIZ], *filename;

⑥ int   rms_status;
```

Key to Example 12–1:

- ① The *rms* module defines the RMS data structures. The *stdio* module contains the Standard I/O definitions. The *ssdef* module contains the system services definitions.
- ② Preprocessor variables and macros are defined. A default file extension `.DAT` is defined.
The sizes of the fields in the record are also defined. Some (such as the social security number field) are given a constant length. Others (such as the record size) are defined as macros; the size of the field is determined with the `sizeof` operator. VAX C evaluates constant expressions, such as `KEY_SIZE`, at compile time. No special code is necessary to calculate this value.
- ③ Static storage for the RMS data structures is declared. The file access block, record access block, and extended attribute block types are defined by the RMS module. One extended attribute block is defined for the primary key and one is defined for the alternate key.
- ④ The records in the file are defined using a structure with four fields of character arrays.
- ⑤ The `BUFSIZ` constant is used to define the size of the array that will be used to buffer input from the terminal. The file-name variable is defined as a pointer to **char**.
- ⑥ The variable `rms_status` is used to receive RMS return status information. After each function call, RMS returns status information as an integer. This return status is used to check for specific errors, end-of-file, or successful program execution.

The main function, shown in Example 12-2, controls the general flow of the program.

Example 12-2: Main Program Section

```
/* This segment of RMSEXP.C contains the main function   *
 * and controls the flow of the program.                 */

① main(argc,argv)
   int  argc;
   char **argv;
   {
②   if (argc < 1 || argc > 2)
       printf("RMSEXP - incorrect number of arguments");
       else
       {
           printf("RMSEXP - Personnel Database \
Manipulation Example\n");

③           filename = (argc == 2 ? **++argv : "personnel.dat");
④           initialize(filename);
⑤           open_file();
           for(;;)
⑥           {
               printf("\nEnter option (A,D,P,T,U) or \
? for help :");
               gets(response);
               if (feof(stdin))
                   break;
               printf("\n\n");

⑦           switch(response[0])
               {
                   case 'a': case 'A':  add_employee();
                                       break;
                   case 'd': case 'D':  delete_employee();
                                       break;
                   case 'p': case 'P':  print_employees();
                                       break;
                   case 't': case 'T':  type_employees();
                                       break;
                   case 'u': case 'U':  update_employee();
                                       break;
                   default:              printf("RMSEXP - \
Unknown Operation.\n");
```

(continued on next page)

Example 12–2 (Cont.): Main Program Section

```
        case '?': case '\0':
            type_options();
        }
    }
}

8   rms_status = sys$close(&fab);
9   if (rms_status != RMS$_NORMAL)
    error_exit("$CLOSE");
}
```

Key to Example 12–2:

- ❶ The main function is entered with two parameters. The first is the number of arguments used to call the program; the second is a pointer to the first argument (file name).
- ❷ This statement checks that you used the correct number of arguments when invoking the program.
- ❸ If a file name is included in the command line to execute the program, that file name is used. If a file extension is not given, .DAT is the file extension. If no file name is specified, then the file name is PERSONNEL.DAT.
- ❹ The file access block, record access block, and extended attribute blocks are initialized.
- ❺ The file is opened using the RMS sys\$open function.
- ❻ The program displays a menu and checks for end-of-file (the character CTRL/Z).
- ❼ A **switch** statement and a set of **case** statements control the function to be called, determined by the response from the terminal.
- ❽ The program ends when CTRL/Z is entered in response to the menu. At that time, the RMS sys\$close function closes the employee file.
- ❾ The rms_status variable is checked for a return status of RMS\$_NORMAL. If the file is not closed successfully, then the error-handling function terminates the program.

Example 12–3 shows the function that initializes the RMS data structures. See the RMS documentation for more information about the file access block, record access block, and extended attribute block structure members.

Example 12-3: Function Initializing RMS Data Structures

```
/* This segment of RMSEXP.C contains the function that      *
 * initializes the RMS data structures.                      */
initialize(char *fn);
{
  ❶ fab = cc$rms_fab; /* Initialize FAB */
  fab.fab$b_bks = 4;
  fab.fab$l_dna = DEFAULT_FILE_NAME;
  fab.fab$b_dns = sizeof DEFAULT_FILE_NAME -1;
  fab.fab$b_fac = FAB$M_DEL | FAB$M_GET |
  FAB$M_PUT | FAB$M_UPD;
  fab.fab$l_fna = fn;
  fab.fab$b_fns = strlen(fn);
  ❷ fab.fab$l_fop = FAB$M_CIF;
  fab.fab$w_mrs = RECORD_SIZE;
  fab.fab$b_org = FAB$C_IDX;
  ❸ fab.fab$b_rat = FAB$M_CR;
  fab.fab$b_rfm = FAB$C_FIX;
  fab.fab$b_shr = FAB$M_NIL;
  fab.fab$l_xab = &primary_key;
  ❹ rab = cc$rms_rab; /* Initialize RAB */
  rab.rab$l_fab = &fab;
  ❺ primary_key = cc$rms_xabkey; /* Initialize Primary *
  * Key XAB */
  primary_key.xab$b_dtp = XAB$C_STG;
  primary_key.xab$b_flg = 0;
  ❻ primary_key.xab$w_pos0 = (char *) &record.ssn -
  (char *) &record;
  primary_key.xab$b_ref = 0;
  primary_key.xab$b_siz0 = SIZE_SSN;
  primary_key.xab$l_nxt = &alternate_key;
  primary_key.xab$l_knm = "Employee Social Security \
Number ";
  ❼ alternate_key = cc$rms_xabkey; /* Initialize Alternate *
  * Key XAB */
  alternate_key.xab$b_dtp = XAB$C_STG;
  ❽ alternate_key.xab$b_flg = XAB$M_DUP | XAB$M_CHG;
  alternate_key.xab$w_pos0 = (char *) &record.last_name -
  (char *) &record;
  alternate_key.xab$b_ref = 1;
  alternate_key.xab$b_siz0 = SIZE_LNAME;
  ❾ alternate_key.xab$l_knm = "Employee Last Name \
";
}
```

Key to Example 12-3:

- ❶ The prototype `cc$rms_fab` initializes the file access block with default values. Some members have no default values; they must be initialized.

Such members include the file-name string address and size. Other members can be initialized to override the default values.

- ② This statement initializes the file-processing options member with the create-if option. A file is created if one does not exist.
- ③ This statement initializes the record attributes member with the carriage-return control attribute. Records are terminated with a carriage return/line feed when they are printed on the printer or displayed at the terminal.
- ④ The prototype `cc$rms_rab` initializes the record access block with the default values. In this case, the only member that must be initialized is the `rab$l_fab` member, which associates a file access block with a record access block.
- ⑤ The prototype `cc$rms_xabkey` initializes an extended attribute block for one key of an indexed file.
- ⑥ The position of the key is specified by subtracting the offset of the member from the base of the structure.
- ⑦ A separate extended attribute block is initialized for the alternate key.
- ⑧ This statement specifies that more than one alternate key can contain the same value (`XAB$M_DUP`), and that the value of the alternate key can be changed (`XAB$M_CHG`).
- ⑨ The key-name member is padded with blanks because it is a fixed-length, 32-character field.

Example 12–4 shows the internal functions for the program.

Example 12-4: Internal Functions

```
/* This segment of RMSEXP.C contains the functions that *
 * control the data manipulation of the program. */

open_file()
{
1  rms_status = sys$create(&fab);
   if (rms_status != RMS$_NORMAL &&
       rms_status != RMS$_CREATED)
       error_exit("$OPEN");

   if (rms_status == RMS$_CREATED)
       printf("[Created new data file.]\n");

2  rms_status = sys$connect(&rab);
   if (rms_status != RMS$_NORMAL)
       error_exit("$CONNECT");
}

3 type_options()
{
   printf("Enter one of the following:\n\n");
   printf("A   Add an employee.\n");
   printf("D   Delete an employee specified by SSN.\n");
   printf("P   Print employee(s) by ascending SSN on \
line printer.\n");

   printf("T   Type employee(s) by ascending last name \
on terminal.\n");
   printf("U   Update employee specified by SSN.\n\n");
   printf("?   Type this text.\n");
   printf("^Z  Exit this program.\n\n");
}

4 pad_record()
{
   int      i;

   for(i = strlen(record.ssn); i < SIZE_SSN; i++)
       record.ssn[i] = ' ';
   for(i = strlen(record.last_name); i < SIZE_LNAME; i++)
       record.last_name[i] = ' ';
   for(i = strlen(record.first_name); i < SIZE_FNAME; i++)
       record.first_name[i] = ' ';
   for(i = strlen(record.comments); i < SIZE_COMMENTS; i++)
       record.comments[i] = ' ';
}

/* This subroutine is the fatal error-handling routine. */
```

(continued on next page)

Example 12-4 (Cont.): Internal Functions

```
⑤ error_exit(operation)
char *operation;
{
    printf("RMSEXP - file %s failed (%s)\n",
          operation, filename);
    exit(rms_status);
}
```

Key to Example 12-4:

- ① The `open_file` function uses the RMS `sys$create` function to create the file, giving the address of the file access block as an argument. The function returns status information to the `rms_status` variable.
- ② The RMS `sys$connect` function associates the record access block with the file access block.
- ③ The `type_options` function, called from the main function, prints help information. Once the help information is displayed, control returns to the main function, which processes the response that is typed at the terminal.
- ④ For each field in the record, the `pad_record` function fills the remaining bytes in the field with blanks.
- ⑤ This function handles fatal errors. It prints the function that caused the error, returns a VMS error code (if appropriate), and exits the program.

Example 12-5 shows the function that adds a record to the file. This function is called when 'a' or 'A' is entered in response to the menu.

Example 12-5: Utility Function: Adding Records

```
/* This segment of RMSEXP.C contains the function that      *
 * adds a record to the file.                               */
add_employee()
{
  ❶ do
    {
      printf("(ADD)  Enter Social Security Number \");
      gets(&response);
    }
    while(strlen(response) == 0);
    strncpy(record.ssn, response, SIZE_SSN);
    do
      {
        printf("(ADD)  Enter Last Name \");
        gets(response);
      }
      while(strlen(response) == 0);
      strncpy(record.last_name, response, SIZE_LNAME);
    do
      {
        printf("(ADD)  Enter First Name \");
        gets(response);
      }
      while(strlen(response) == 0);
      strncpy(record.first_name, response, SIZE_FNAME);
    do
      {
        printf("(ADD)  Enter Comments \");
        gets(response);
      }
      while(strlen(response) == 0);
      strncpy(record.comments, response, SIZE_COMMENTS);
  ❷ pad_record();
  ❸ rab.rab$b_rac = RAB$C_KEY;
    rab.rab$l_rbf = &record;
    rab.rab$w_rsz = RECORD_SIZE;
}
```

(continued on next page)

Example 12-5 (Cont.): Utility Function: Adding Records

```
④ rms_status = sys$put(&rab);
⑤ if (rms_status != RMS$_NORMAL && rms_status !=
    RMS$_DUP && rms_status != RMS$_OK_DUP)
    error_exit("$PUT");
    else
        if (rms_status == RMS$_NORMAL || rms_status ==
            RMS$_OK_DUP)
            printf("[Record added successfully.]\n");
        else
            printf("RMSEXP - Existing employee with same SSN, \
not added.\n");
    }
```

Key to Example 12-5:

- ① A series of **do** loops controls the input of information. For each field in the record, a prompt is displayed. The response is buffered and the field is copied to the structure.
- ② When all fields have been entered, the `pad_record` function pads each field with blanks.
- ③ Three members in the record access block are initialized before writing the record. The record access member (`rab$b_rac`) is initialized for keyed access. The record buffer and size members (`rab$l_rbf` and `rab$w_rsz`) are initialized with the address and size of the record to be written.
- ④ The RMS `sys$put` function writes the record to the file.
- ⑤ The `rms_status` variable is checked. If the return status is normal, or if the record has a duplicate key value and duplicates are allowed, the function prints a message stating that the record was added to the file. Any other return value is treated as a fatal error causing `error_exit` to be called.

Example 12-6 shows the function that deletes records. This function is called when 'd' or 'D' is entered in response to the menu.

Example 12–6: Utility Function: Deleting Records

```
/* This segment of RMSEXP.C contains the function that      *
 * deletes a record from the file.                          */
delete_employee()
{
    ❶ int i;
    do
    {
        printf("(DELETE) Enter Social Security Number  ");
        gets(response);
        i = strlen(response);
    }
    while(i == 0);
    ❷ while(i < SIZE_SSN)
        response[i++] = ' ';
    ❸ rab.rab$b_krf = 0;
    rab.rab$l_kbf = &response;
    rab.rab$b_ksz = SIZE_SSN;
    rab.rab$b_rac = RAB$C_KEY;
    ❹ rms_status = sys$find(&rab);
    ❺ if (rms_status != RMS$NORMAL && rms_status != RMS$RNF)
        error_exit("$FIND");
    else
        if (rms_status == RMS$RNF)
            printf("RMSEXP - specified employee does not \
            exist.\n");
        else
        {
            ❻ rms_status = sys$delete(&rab);
            if (rms_status != RMS$NORMAL)
                error_exit("$DELETE");
        }
}
```

Key to Example 12–6:

- ❶ A **do** loop prompts you to type a social security number at the terminal and places the response in the response buffer.
- ❷ The social security number is padded with blanks.
- ❸ Some members in the record access block must be initialized before the program can locate the record. Here, the key of reference (0 specifies the primary key), the location and size of the search string (this is the address of the response buffer and its size), and the type of record access (in this case, keyed access) are given.

- ④ The RMS `sys$find` function locates the record specified by the social security number entered from the terminal.
- ⑤ The program checks the `rms_status` variable for the values `RMS$_NORMAL` and `RMS$_RNF` (record not found). A message is displayed if the record cannot be found. Any other error is a fatal error.
- ⑥ The RMS `sys$delete` function deletes the record. The return status is checked only for success.

The `type_employees` function in Example 12-7 displays the employee file at the terminal. This function is called from the main function when 't' or 'T' is entered in response to the menu.

Example 12-7: Utility Function: Typing the File

```

/* This segment of RMSEXP.C contains the function that      *
 * displays a single record at the terminal.                  */
type_employees()
{
  ① int number_employees;
  ② rab.rab$b_krf = 1;
  ③ rms_status = sys$rewind(&rab);
    if (rms_status != RMS$_NORMAL)
        error_exit("$REWIND");
  ④ printf("\n\nEmployees (Sorted by Last Name)\n\n");
    printf("Last Name      First Name      SSN          \
           Comments\n");
    printf("-----      -----      -----\
           -----\n\n");
  ⑤ rab.rab$b_rac = RAB$_SEQ;
    rab.rab$l_ubf = &record;
    rab.rab$w_usz = RECORD_SIZE;
  ⑥ for(number_employees = 0; ; number_employees++)
    {
        rms_status = sys$get(&rab);
        if (rms_status != RMS$_NORMAL && rms_status !=
            RMS$_EOF)
            error_exit("$GET");
        else
            if (rms_status == RMS$_EOF)
                break;
    }
}

```

(continued on next page)

Example 12-7 (Cont.): Utility Function: Typing the File

```
        printf("%.*s%.*s%.*s%.*s\n",
              SIZE_LNAME, record.last_name,
              SIZE_FNAME, record.first_name,
              SIZE_SSN, record.ssn,
              SIZE_COMMENTS, record.comments);
    }
7  if (number_employees)
    printf("\nTotal number of employees = %d.\n",
          number_employees);
    else
    printf("[Data file is empty.]\n");
}
```

Key to Example 12-7:

- ① A running total of the number of records in the file is kept in the `number_employees` variable.
- ② The key of reference is changed to the alternate key so that the employees are displayed in alphabetical order by last name.
- ③ The file is positioned to the beginning of the first record according to the new key of reference, and the return status of the `sys$rewind` function is checked for success.
- ④ A heading is displayed.
- ⑤ Sequential record access is specified, and the location and size of the record is given.
- ⑥ A **for** loop controls the following operations:
 - Incrementing the `number_employees` counter
 - Locating a record and placing it in the record structure, using the `RMS sys$get` function
 - Checking the return status of the `RMS sys$get` function
 - Displaying the record at the terminal
- ⑦ This **if** statement checks for records in the file. The result is a display of the number of records or a message indicating that the file is empty.

Example 12-8 shows the function that prints the file on the printer. This function is called by the main function when 'p' or 'P' is entered in response to the menu.

Example 12–8: Utility Function: Printing the File

```
/* This segment of RMSEXP.C contains the function that *
 * prints the file. */
print_employees()
{
    int number_employees;
    FILE *fp;
    ❶ fp = fopen("personnel.lis", "w", "rat=cr",
                "rfm=var", "fop=spl");
    if (fp == NULL)
    {
        perror("RMSEXP - failed opening listing \
file");
        exit(SS$NORMAL);
    }
    ❷ rab.rab$b_krf = 0;
    ❸ rms_status = sys$rewind(&rab);
    if (rms_status != RMS$NORMAL)
        error_exit("$REWIND");
    ❹ fprintf(fp, "\n\nEmployees (Sorted by SSN)\n\n");
    fprintf(fp, "Last Name      First Name      SSN          \
Comments\n");
    fprintf(fp, "-----      -----      -----\
-----\n\n");
    ❺ rab.rab$b_rac = RAB$C_SEQ;
    rab.rab$l_ubf = &record;
    rab.rab$w_usz = RECORD_SIZE;
    ❻ for(number_employees = 0; ; number_employees++)
    {
        rms_status = sys$get(&rab);
        if (rms_status != RMS$NORMAL &&
            rms_status != RMS$EOF)
            error_exit("$GET");
        else
            if (rms_status == RMS$EOF)
                break;
    }
}
```

(continued on next page)

Example 12-8 (Cont.): Utility Function: Printing the File

```
        fprintf(fp, "%. *s%. *s%. *s%. *s%",
                SIZE_LNAME, record.last_name,
                SIZE_FNAME, record.first_name,
                SIZE_SSN, record.ssn,
                SIZE_COMMENTS, record.comments);
    }
7   if (number_employees)
        fprintf(fp, "Total number of employees = %d.\n",
                number_employees);
    else
        fprintf(fp, "[Data file is empty.]\n");
8   fclose(fp);
        printf("[Listing file \"personnel.lis\" spooled to \
SYS$PRINT.]\n");
    }
```

Key to Example 12-8:

- ① This function creates a sequential file with carriage-return-control, variable-length records. It spools the file to the printer when the file is closed. The file is created using the Standard I/O Run-Time Library function **fopen**, which associates the file with the file pointer, **fp**.
- ② The key of reference for the indexed file is the primary key.
- ③ The RMS **sys\$rewind** function positions the file at the first record. The return status is checked for success.
- ④ A heading is written to the sequential file using the Standard I/O function **fprintf**.
- ⑤ The record access, user buffer address, and user buffer size members of the record access block are initialized for keyed access to the record located in the record structure.
- ⑥ A **for** loop controls the following operations:
 - Initializing the running total and then incrementing the total at each iteration of the loop
 - Locating the records and placing them in the record structure with the RMS **sys\$get** function, one record at a time
 - Checking the **rms_status** information for success and end-of-file
 - Writing the record to the sequential file
- ⑦ The **number_employees** counter is checked. If it is 0, a message is printed indicating that the file is empty. If it is not 0, the total is printed at the bottom of the listing.

- ⑧ The sequential file is closed. Since it has the spl record attribute, the file is automatically spooled to the printer. The function displays a message at the terminal stating that the file was successfully spooled.

Example 12-9 shows the function that updates the file. This function is called by the main function when 'u' or 'U' is entered in response to the menu.

Example 12-9: Utility Function: Updating the File

```
/* This segment of RMSEXP.C contains the function that      *
 * updates the file.                                       */
update_employee()
{
  ①  int i;
     do
     {
       printf("(UPDATE) Enter Social Security Number\
");
       gets(response);
       i = strlen(response);
     }
     while(i == 0);
  ②  while(i < SIZE_SSN)
     response[i++] = ' ';
  ③  rab.rab$b_krf = 0;
     rab.rab$l_kbf = &response;
     rab.rab$b_ksz = SIZE_SSN;
     rab.rab$b_rac = RAB$C_KEY;
     rab.rab$l_ubf = &record;
     rab.rab$w_usz = RECORD_SIZE;
  ④  rms_status = sys$get(&rab);
     if (rms_status != RMS$_NORMAL && rms_status != RMS$_RNF)
       error_exit("$GET");
     else
       if (rms_status == RMS$_RNF)
         printf("RMSEXP - specified employee does not \
exist.\n");
  ⑤  else
     {
       printf("Enter the new data or RETURN to leave \
data unmodified.\n\n");
     }
}
```

(continued on next page)

Example 12–9 (Cont.): Utility Function: Updating the File

```
printf("Last Name:");
gets(response);
if (strlen(response))
    strncpy(record.last_name, response,
            SIZE_LNAME);

printf("First Name:");
gets(response);
if (strlen(response))
    strncpy(record.first_name, response,
            SIZE_FNAME);

printf("Comments:");
gets(response);
if (strlen(response))
    strncpy(record.comments, response,
            SIZE_COMMENTS);

6   pad_record();
7   rms_status = sys$update(&rab);
    if (rms_status != RMS$_NORMAL)
        error_exit("$UPDATE");

    printf("[Record has been successfully \
updated.]\n");
    }
}
```

Key to Example 12–9:

- 1 A **do** loop prompts for the social security number and places the response in the response buffer.
- 2 The response is padded with blanks so that it will correspond to the field in the file.
- 3 Some of the members in the record access block are initialized for the operation. The primary key is specified as the key of reference, the location and size of the key value are given, keyed access is specified, and the location and size of the record are given.
- 4 The RMS `sys$get` function locates the record and places it in the record structure. The function checks the `rms_status` value for `RMS$_NORMAL` and `RMS$_RNF` (record not found). If the record is not found, a message is displayed. If the record is found, the program prints instructions for updating the record.

- ⑤ For each field (except the social security number, which cannot be changed), the program displays the current value for that field. If you press the RETURN key, the record is placed in the record structure unchanged. If you make a change to the record, the new information is placed in the record structure.
- ⑥ The fields in the record are padded with blanks.
- ⑦ The RMS sys\$update function rewrites the record. The program then checks that the update operation was successful. Any error causes the program to call the fatal error-handling routine.

Using VAX C in the Common Language Environment

The VAX C compiler is part of the VMS common language environment. This environment defines certain calling procedures and guidelines that allow you to call routines written in different languages from VAX C programs, to call VAX C functions from programs written in other languages, or to call prewritten system routines from VAX C programs. You can call any one of the following routine types from VAX C:

- Routines written in other VAX languages
- VMS RTL routines
- VMS system services
- VMS utility routines

The terms *routine*, *procedure*, and *function* are used throughout this chapter. A *routine* is a closed, ordered set of instructions that performs one or more specific tasks. Every routine has an entry point (the routine name), and optionally an argument list. Procedures and functions are specific types of routines: a *procedure* is a routine that does not return a value; a *function* is a routine that returns a value by assigning that value to the function's identifier.

System routines are prewritten VMS routines that perform common tasks, such as finding the square root of a number or allocating virtual memory. You can call any system routine from your program, provided that VAX C supports the data structures required to call the routine. The system routines used most often are VMS RTL routines and system services. System routines, which are discussed later in this chapter, are documented in detail in the *VMS Run-Time Library Routines Volume* and the *VMS System Services Reference Manual*.

This chapter discusses the following topics:

- The VAX Procedure Calling and Condition Handling Standard (Section 13.1)
- Specifying parameter-passing mechanisms (Section 13.2)
- VAX C default parameter-passing mechanisms (Section 13.2.4)
- Interlanguage calling (Section 13.3)
- VMS RTL routines (Section 13.5)
- VMS system services routines (Section 13.6)
- Calling routines (Section 13.7)
- Variable-length argument lists in system services (Section 13.8)
- Return status values (Section 13.9)
- Examples of calling system routines (Section 13.10)

13.1 The VAX Procedure Calling and Condition Handling Standard

The VAX Procedure Calling and Condition Handling Standard describes the concepts used by all VAX languages to invoke routines and pass data between them. The following attributes are specified by the VAX Procedure Calling and Condition Handling Standard:

- Register usage
- Stack usage
- Function return value
- Argument list

The following sections discuss these attributes in more detail. The VAX Procedure Calling and Condition Handling Standard also defines such attributes as the calling sequence, the argument data types and descriptor formats, condition handling, and stack unwinding. These attributes are discussed in detail in the *Introduction to VMS System Routines*.

13.1.1 Register and Stack Usage

The VAX Procedure Calling and Condition Handling Standard defines several registers and their uses, as listed in Table 13–1.

Table 13–1: VAX Register Usage

Register	Use
PC	Program counter
SP	Stack pointer
FP	Current stack frame pointer
AP	Argument pointer
R1	Environment value (when necessary)
R0, R1	Function return value registers

By definition, any called routine can use registers R2 through R11 for computation, and the AP register as a temporary register.

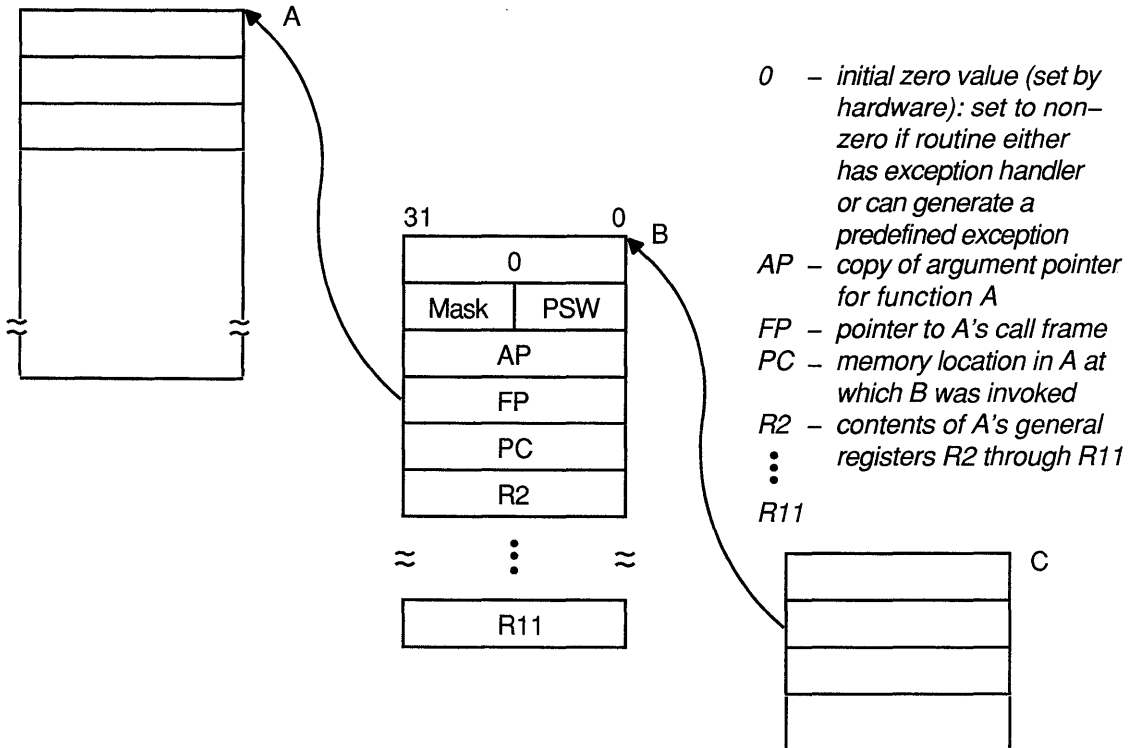
In the VAX Procedure Calling and Condition Handling Standard, a *stack* is defined as a last-in/first-out (LIFO) temporary storage area that the system allocates for every user process. The system keeps information about each routine call in the current image on the call stack. Then, each time you call a routine, the system creates a structure on this call stack, known as the *call frame*. The call frame for each active process contains the following data:

- A pointer to the call frame of the previous routine call. This pointer corresponds to the frame pointer (FP).
- The argument pointer (AP) of the previous routine call.
- The storage address of the point at which the routine was called; that is, the address of the instruction following the call to the current routine. This address is called the program counter (PC).
- The contents of other general registers. Based on a mask specified in the control information, the system restores the saved contents of these registers to the calling routine when control returns to it.

When a routine completes execution, the system uses the frame pointer in the call frame of the current routine to locate the frame of the previous routine. The system then removes the call frame of the current routine from the stack.

Figure 13–1 shows the call stack and several call frames. Function A calls function B, which calls function C. When a function reaches a **return** statement or when control reaches the end of the function, the system uses the frame pointer in the call frame of the current function to locate the frame of the previous function. It then removes the call frame of the current function from the stack.

Figure 13–1: The Call Stack



ZK-0090-GE

13.1.2 Return of the Function Value

A function is a routine that returns a single value to the calling routine. The *function value* represents the value of the expression in the **return** statement. According to the VAX Procedure Calling and Condition Handling Standard, a function value may be returned as either an actual value or a condition value that indicates success or failure.

13.1.3 The Argument List

The VAX Procedure Calling and Condition Handling Standard also defines a data structure called the argument list. An *argument list* is a collection of longwords in memory that represents a routine parameter list and possibly includes a function value. You use an argument list to pass information to a routine and receive results. Figure 13–2 shows the structure of a typical argument list.

Figure 13–2: Structure of a VAX Argument List

0	n
arg1	
arg2	
.	
.	
.	
argn	

ZK–5503–GE

The first longword must be present; this longword stores the number of arguments (the argument count: *n*) as an unsigned integer value in the low byte of the longword with a maximum of 255 arguments. The remaining 24 bits of the first longword are reserved for use by DIGITAL and should be 0.

The longwords labeled *arg1* through *argn* are the actual parameters, which can be any of the following addresses or value:

- An uninterpreted 32-bit value that is passed by value
- An address that is passed by reference
- An address of a descriptor that is passed by descriptor

The argument list contains the parameters that are passed to the routine. Depending on the passing mechanisms for these parameters, the forms of the arguments contained in the argument list vary. For example, if you pass three arguments, the first by value, the second by reference, and the third by descriptor, the argument list would contain the value of the first argument, the address of the second, and the address of the descriptor of the third. Figure 13-3 shows this argument list.

Figure 13-3: Example of a VAX Argument List

0	3
value of the first parameter	
address of the second parameter	
address of descriptor of the third parameter	

ZK-5504-GE

For additional information on the VAX Procedure Calling and Condition Handling Standard, see the *Introduction to VMS System Routines*.

13.2 Specifying Parameter-Passing Mechanisms

When you pass data between routines that are not written in the same VAX language, you have to specify how you want that data to be represented and interpreted. You do this by specifying a *parameter-passing mechanism*.

The calling standard defines three ways to pass data in an argument list. When you code a reference to a non-VAX C procedure, you must know how to pass each argument and write the function reference accordingly.

The following list describes the three argument-passing mechanisms:

- **By immediate value**
When an argument is passed by immediate value, the actual value of the argument is present in the argument list. This is the default argument-passing mechanism for all function references written in VAX C.
- **By reference**
When an argument is passed by reference, the address of the argument is present in the argument list. Use the VAX C ampersand operator (&) to pass the address of an argument, or pass a pointer to the argument by value.
- **By descriptor**
When an argument is passed by descriptor, the address of a data structure describing the argument is present in the argument list. From a VAX C program, you pass a descriptor first by creating a structure (**struct**) that meets the descriptor requirements of the called procedure and then by passing the structure's address with the ampersand operator or by passing a pointer to that structure by value.

NOTE

In the C programming language environment, you can take the address of an argument and use that address to access the values of subsequent arguments in that argument list, operating on the assumption that the compiler did not propagate any of the arguments to registers. This is possible using the current implementation of VAX C.

However, accessing an argument list is *not* an advisable practice in the VMS environment under the VAX Calling Standard. Also, accessing argument lists in this manner is not portable and may not be possible in future releases of VAX C. For an alternate method of accessing variable-length argument lists in the VMS environment, see LIB\$CALLG in the *VMS Run-Time Library Routines Volume*.

The following sections outline each of these parameter-passing mechanisms in more detail.

13.2.1 Passing Arguments by Immediate Value

By default, all values or expressions in a VAX C function's argument list are passed by immediate value. The expressions are evaluated and the results placed directly in the argument list of the CALL machine instruction.

The following statement declares the entry point of the Set Event Flag SYS\$SETEF system service, which is used to set a specific event flag to 1:

```
/* Declare the function as a function returning type int.      */
int  SYS$SETEF();
```

The Set Event Flag system service call requires one argument—the number of the event flag to be set—to be passed by immediate value. VAX C converts linker-resolved variable names (such as the entry-point names of system service calls) to uppercase. You do not have to declare them in uppercase in your program. However, linker-resolved variable names must be declared and used with identical cases in each module. The documentation uses uppercase as a convention for referring to system service calls to highlight them in the text and examples.

Like all system services, SYS\$SETEF returns an integer value (the return status of the service) in register 0. Most system services return an integer completion status; therefore, the system service does not always have to be declared before it is used. The examples in this chapter declare system services for completeness.

VAX C does not require you to declare a function or to specify the number or types of the function's arguments. However, if you call a function without declaring it or without providing argument information in the declaration, VAX C does not check the types of the arguments in a call to that function. If you declare a function prototype, the compiler does check the arguments in a call to make sure that they have the same type. (See Section 5.3 for more information on function prototypes.)

In the *VMS System Services Volume*, you can find the specification of each service's arguments. SYS\$SETEF, for example, takes one argument, an event flag number. It returns one of four status values, which are represented by the fsymbolic constants shown in Table 13-2.

Table 13–2: Status Values of SYS\$SETEF

Returned	Status	Description
SS\$_WASCLR	Success	Flag was previously clear
SS\$_WASSET	Success	Flag was previously set
SS\$_ILLEFC	Failure	Illegal event flag number
SS\$_UNASEFC	Failure	Event flag not in associated cluster

The system services manual also defines event flags as integers in the range 0 to 127, grouped in clusters of 32. Clusters 0 and 1, comprising flags 0 to 31 and 32 to 63, respectively, are local clusters available to any process, with the restriction that flags 24 to 31 are reserved for use by the VMS system. There are many ways of passing valid event flag numbers from your VAX C program to SYS\$SETEF. One way is to use **enum** to define a subset of integers, as follows:

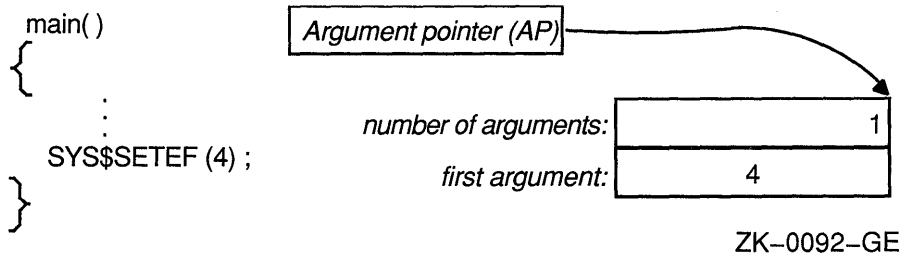
```
enum cluster0 {completion, breakdown, beginning} event;
```

After the flag numbers are defined, call the SYS\$SETEF service with the following code:

```
.  
. .  
int status;  
event = completion;  
. .  
status = SYS$SETEF(event);          /* Set event flag          */  
. .  
.
```

Figure 13–4 shows an argument being passed by immediate value; in this case, the event flag number passed to SYS\$SETEF.

Figure 13-4: Passing Arguments by Immediate Value



Since argument lists consist of longwords, the calling standard dictates that immediate-value arguments be expressed in 32 bits. A single-precision, floating-point (F_floating) value is only 32 bits long, but all arguments of type **float** are promoted by VAX C to **double** (64 bits on a VAX) unless a function prototype declaration is used for the called function. This double-precision value is passed as two immediate values (two longwords).

NOTE

The passing of double-precision immediate values is a violation of the VAX Calling Standard, but is an allowed exception for VAX C.

On rare occasions, the **float-to-double** promotion requires some additional programming. For instance, the function OTS\$POWRJ, in the VAX Common Run-Time Procedure Library, computes the value of a floating-point number raised to the power of a signed longword (in VAX C terms, a **float** to the power of an **int**). This function (and others like it) is called implicitly by high-level VAX languages that have an exponentiation operator as part of the language. It requires that both its arguments be passed as immediate values, and it returns a single-precision (**float**) result. To pass a floating-point base to the procedure, you must use some method to avoid promoting **float** arguments. The recommended method is to declare the procedure using a function prototype declaration, as shown in Example 13-1.

Example 13-1: Passing Floating-Point Arguments by Immediate Value

```
/* This program shows how to pass a floating-point value,      *
 * using prototypes to avoid promoting floating                *
 * arguments to arguments of type double.                      */
#include stdio

/* This declared function returns a value of type float.  It  *
 * should be called as follows: OTS$POWRJ(base, power),      *
 * where base is of type float and power is of type int.     */
float OTS$POWRJ(float, int);

main()
{
    /* To hold result of                                     *
     * OTS$POWRJ                                             */
    float result;
    int power; /* Power argument */

    float base;

    base = 3.145; /* Assign constant to base */
    power = 2;
    result = OTS$POWRJ(base, power);

    printf("Result= %f\n", result);
}
```

Most run-time functions that operate on floating-point values take their arguments by reference, so a prototype is not usually necessary. In addition, the example does not show the methods for handling arithmetic errors that result from the operation performed. For more information on error handling in this context, and on the run-time library in general, see the *VMS Run-Time Library Routines Volume*.

When you pass a parameter by value, you pass a copy of the parameter value to the routine instead of passing its address. Because the actual value of the parameter is passed, the routine does not have access to the storage location of the parameter; therefore, any changes that you make to the parameter value in the routine do not affect the value of that parameter in the calling routine.

13.2.2 Passing Arguments by Reference

Some system services and run-time library procedures expect arguments passed by reference. This means that the argument list contains the address of the argument rather than its value. This mechanism is also used by default by some programming languages, such as PL/I, and is available as an option in others, such as Pascal.

In VAX C, you can use the ampersand operator (&) to pass an argument by reference; that is, the ampersand operator causes the argument's address to be passed. Note that an array name without brackets or a function name without parentheses in an argument list always results in passing the address of the array or function; the ampersand is unnecessary. You can also pass a pointer by value, which is the same as passing the item it points to by reference.

In the special case of argument lists, VAX C allows the ampersand operator to be used on constants as well. You should limit this use of the ampersand solely to calls to VMS system functions to ensure portability of your VAX C programs to other C compilers.

For example, the Read Event Flags (SYS\$READEF) system service requires that its first argument be passed by immediate value and its second argument be passed by reference. SYS\$READEF returns the status of all the event flags in a particular cluster. (Event flags are numbered from 0 to 127 and arranged in clusters of 32, such that flags 0 to 31 comprise cluster 0, flags 32 to 63, cluster 1, and so forth.)

The first SYS\$READEF argument is any event flag number in the cluster of interest. The second argument is the address of a longword that receives the status of all 32 event flags in that cluster. In addition to the event-flag status value, the system service returns one of the status values shown in Table 13-3 expressed as a global symbol.

Table 13-3: Status Values of SYS\$READEF

Returned	Status	Description
SS\$_WASCLR	Success	Specified event flag was clear
SS\$_WASSET	Success	Specified event flag was set
SS\$_ACCVIO	Failure	Could not write to status longword
SS\$_ILLEFC	Failure	Event flag number was illegal
SS\$_UNASEFC	Failure	Cluster of interest not accessible

Example 13-2 shows a call to the SYS\$READEF system service from a VAX C program.

Example 13-2: Passing Arguments by Reference

```
/* This program shows how to call system service SYS$READEF. */
#include ssdef
#include stdio
int SYS$READEF();
main()
{
    /* Longword that receives *
    * the status of the *
    * event flag cluster */
    unsigned cluster_status;

    int return_status; /* Status: SYS$READEF */

    /* Argument values for *
    * SYS$READEF */
    enum cluster0
    {
        completion, breakdown, beginning
    } event;
    .
    .
    .
    event = completion; /* Event flag in cluster 0 */

    /* Obtain status of *
    * cluster 0. Pass value *
    * of event and address *
    * of cluster_status. */
    return_status = SYS$READEF(event, &cluster_status);

    /* Check for successful *
    * call */
    if (return_status != SS$WASCLR && return_status != SS$WASSET)
    {
        /* Handle the error here. */
        .
        .
        .
    }
    else
    {
        /* Check bits of interest in cluster_status here. */
        .
        .
        .
    }
}
```

13.2.3 Passing Arguments by Descriptor

A *descriptor* is a structure that describes the data type, size, and address of a data structure. According to the VAX Calling Standard, you must pass a descriptor by placing its address in the argument list. To pass an argument by descriptor from a VAX C program, perform the following steps:

1. Write a structure declaration that models the required descriptor. This involves including the *descrip* text-library module to define **struct** tags for all the forms of descriptors.
2. Assign appropriate values to the structure members.
3. Use the structure name, with an ampersand operator (&) in the function reference, to put the structure's address in the argument list.

VAX C never passes arguments by descriptor by default; you must take explicit action to pass an argument by descriptor. Also, if you write structure or union names in a function's argument list without the ampersand operator, the structure or union is passed by immediate value to the called function. You pass arguments by descriptor only when the called function is written in another language and explicitly requires this mechanism.

NOTE

The passing of structures as immediate values can be a violation of the VAX Calling Standard if the entire structure is larger than one longword of memory. This type of argument passing is an allowed exception for VAX C.

There are several classes of descriptor. Each class requires that certain bits be set in the first longword of the descriptor. For more information about the descriptors and their formats, see the *Introduction to VMS System Routines*. You can model descriptors in VAX C as follows:

```
struct dsc$descriptor
{
    unsigned short dsc$w_length; /* Length of data      */
    char dsc$b_dtype           /* Data type code     */
    char dsc$b_class          /* Descriptor class   */
    * code                    /* code               */
    char *dsc$a_pointer       /* Has address of first */
    * byte                    /* byte               */
};
```

In this model, the variable `dsc$w_length` is a 16-bit word containing the length of the entire data; the unit (for example, bit or byte) in which the length is measured depends on the descriptor class. The member `dsc$b_dtype` is a byte containing a numeric code; the code denotes the data type of the data. The class member `dsc$b_class` is another byte code giving the descriptor class. Table 13–4 shows the valid class codes.

Table 13–4: Valid Class Codes

Class Code	Symbolic Name	Descriptor Class
1	DSC\$K_CLASS_S	Scalar, string
2	DSC\$K_CLASS_D	Dynamic string descriptor
3	—	Reserved by DIGITAL
4	DSC\$K_CLASS_A	Array
5	DSC\$K_CLASS_P	Procedure
6	DSC\$K_CLASS_PI	Procedure incarnation
7	DSC\$K_CLASS_J	Label
8	DSK\$K_CLASS_JI	Label incarnation
9	DSC\$K_CLASS_SD	Decimal scalar string
10	DSC\$K_CLASS_NCA	Noncontiguous array
11	DSC\$K_CLASS_VS	Varying string
12	DSC\$K_CLASS_VSA	Varying string array
13	DSC\$K_CLASS_UBS	Unaligned bit string
14	DSC\$K_CLASS_UBA	Unaligned bit array
15	DSC\$K_CLASS_SB	String with bounds descriptor
16	DSC\$K_CLASS_UBSB	Unaligned bit string with bounds descriptor
17-190	—	Reserved by DIGITAL
191	DSC\$K_CLASS_BFA	Basic file array
192-255	—	Reserved for customer applications

The atomic data types shown in Table 13–5 are supported by VAX C; all others are not directly supported by the language. See the *Introduction to VMS System Routines* manual for a complete list of atomic class codes.

Table 13–5: Atomic Data Types

Class Code	Symbolic Name	Descriptor Class
2	DSC\$K_DTYPE_BU	Byte (unsigned)
3	DSC\$K_DTYPE_WU	Word (unsigned)
4	DSC\$K_DTYPE_LU	Longword (unsigned)
6	DSC\$K_DTYPE_B	Byte integer (signed)
7	DSC\$K_DTYPE_W	Word integer (signed)
8	DSC\$K_DTYPE_L	Longword integer (signed)
10	DSC\$K_DTYPE_F	F_floating
11	DSC\$K_DTYPE_D	D_floating
27	DSC\$K_DTYPE_G	G_floating

The last member of the structure model, `dsc$a_pointer`, points to the first byte of the data.

To pass an argument by descriptor, you define and assign values to the data following normal VAX C programming practices. You must define a `dsc$descriptor` structure and assign the data's address to the `dsc$a_pointer` member. You must also assign appropriate values to the members `dsc$w_length`, `dsc$b_dtype`, and `dsc$b_class`. For the specific requirements of each descriptor class, see the *Introduction to VMS System Routines*.

For example, the Set Process Name (SYS\$SETPRN) system service, which enables a process to establish or change its process name, accepts a process name as a fixed-length character string passed by descriptor. The character string can have from 1 to 15 characters. The system service returns the status values denoted by the global names `SS$_NORMAL`, `SS$_ACCVIO`, `SS$_DUPLNAM`, and `SS$_IVLOGNAM` (for normal completion, inaccessible descriptor, duplicate process name, and invalid length, respectively).

Example 13–3 shows a call to this system service from a VAX C program.

Example 13-3: Passing Arguments by Descriptor

```
/* This program shows a call to system service SYS$SETPRN. */
#include ssdef
#include stdio
/* Define structures for descriptors */
#include descrip
int SYS$SETPRN();
main()
{
    int ret; /* Define return status of SYS$SETPRN */
    struct dsc$descriptor_s name_desc; /* Name the descriptor */
    char *name = "NEWPROC"; /* Define new process name */
    .
    .
    . /* Length of name WITHOUT null terminator */
    name_desc.dsc$w_length = strlen(name);
    .
    . /* Put address of shortened string in descriptor */
    name_desc.dsc$a_pointer = name;
    .
    . /* String descriptor class */
    name_desc.dsc$b_class = DSC$K_CLASS_S;
    .
    . /* Data type: ASCII string */
    name_desc.dsc$b_dtype = DSC$K_DTYPE_T;
    .
    .
    ret = SYS$SETPRN(&name_desc);
    if (ret != SS$_NORMAL) /* Test return status */
        fprintf(stderr, "Failed to set process name\n"),
        exit(ret);
    .
    .
}
```

In Example 13-3, the call to `SYS$SETPRN` *must* use the ampersand operator; otherwise, `name_desc`, rather than its address, is passed.

Although this example explicitly sets individual fields in its `name_desc` string descriptor, in practice, the run-time initialization of compile-time constant string descriptors is not performed in this manner. Instead, the fields of compile-time constant descriptors are usually initialized with initialized structures of storage class **static**.

For the purpose of string descriptor initialization, VAX C provides a simple preprocessor macro in the *descrip* text-library module. This macro is named `$DESCRIPTOR`. It takes two arguments, which it uses in a standard VAX C structure declaration. The first argument is an identifier specifying the name of the descriptor to be declared and initialized. The second argument is a pointer to the data byte to be used as the value of the descriptor. Since a character-string constant is interpreted as an initialized pointer to **char**, you may specify the second argument as a simple string constant. You may use the `$DESCRIPTOR` macro in any context where a declaration may be used. The scope of the declared string descriptor identifier name is identical to the scope of a simple **struct** definition as expanded by the macro.

Example 13–4 shows a variant of the program in Example 13–3. Here, the `$DESCRIPTOR` macro is used to create a compile-time string descriptor and to pass it to the `SYS$SETPRN` system service routine. In Example 13–4, the program returns the status value returned by `SYS$SETPRN` to DCL for interpretation.

Example 13–4: Passing Compile-Time String Descriptors

```

/* This program returns the status value returned by          *
 * SYS$SETPRN.                                               */
#include descrip                                             /* Define $DESCRIPTOR *
/* macro                                                    */
int SYS$SETPRN();
main()
{
/* Initialize structure *
 * name_desc as string *
 * descriptor          */
    static $DESCRIPTOR(name_desc, "NEWPROC");
    return SYS$SETPRN(&name_desc);
}

```

The `$DESCRIPTOR` macro is used in further examples in this chapter.

13.2.4 VAX C Default Parameter-Passing Mechanisms

There are default parameter-passing mechanisms established for every data type you can use with VAX C. Table 13–6 lists the VAX C data types you can use with each parameter-passing mechanism. Asterisks appear next to the default parameter-passing mechanism for that particular data type.

Table 13–6: Valid Parameter-Passing Mechanisms in VAX C

Data Type	By Reference	By Descriptor	By Value
Numeric data:			
Variables	Yes	Yes	Yes*
Constants	Yes	Yes	Yes*
Expressions	No	No	Yes*
Array elements	Yes	Yes	Yes*
Entire array	Yes*	Yes	No
String constants	Yes*	Yes	No
Structures and unions	Yes	Yes	Yes*
Functions	Yes*	Yes	No

You must use the appropriate parameter-passing mechanisms whenever you call a routine written in some other VAX language or some prewritten system routine.

13.3 Interlanguage Calling

In VAX C, you can call external routines written in other languages or VAX C routines from routines written in other languages as either functions or subroutines. When you call an external routine as a function, a single value is returned. When you call an external routine as a subroutine (a **void** function), values are returned in the argument list.

By default, VAX C passes all arguments by immediate value with the exception of arrays and functions; these are passed by reference. Table 13–7 lists the default passing mechanisms for other VAX-native languages.

Table 13–7: Default Passing Mechanisms

Language	Arrays	Numeric Data	Character Data
MACRO	No default	No default	No default
Pascal	Reference	Reference	Reference
BASIC	Descriptor	Reference	Descriptor
COBOL	N/A	Reference	Reference
FORTRAN	Reference	Reference	Descriptor

The following sections describe the methods involved in using VAX C with routines written in other VAX-native languages.

13.3.1 Calling VAX FORTRAN

When calling VAX FORTRAN from VAX C or vice versa, note these considerations. VAX FORTRAN argument lists and argument descriptors are usually allocated statically. When it is possible, and to optimize space and time, the VAX FORTRAN compiler pools the argument lists and initializes them at compile time. Sometimes several calls may use the same argument list.

In VAX C, you often use arguments as local variables, modifying them at will. If a VAX C routine that modifies an argument is called from a VAX FORTRAN routine, unintended and incorrect side effects may occur.

The following example shows a VAX C routine that is invalid when called from VAX FORTRAN:

```
void f(x)
int *x;          /* A FORTRAN INTEGER passed by reference */
{
    /* The next assignment is OK. It is permitted to modify what a
     * FORTRAN argument list entry points to. */
    *x = 0;      /* ok */

    /* The next assignment is invalid. It is not permitted to modify
     * a FORTRAN argument list entry itself. */
    x = x + 1;  /* Invalid */
}
```

Another problem is the semantic mismatch between strings in VAX C and strings in VAX FORTRAN. Strings in VAX C vary in length and end in a NUL character. Strings in VAX FORTRAN do not end in a NUL character and are padded with spaces to some fixed length. In general, this mismatch means that strings may not be passed between VAX C and VAX FORTRAN

unless you do additional work. You may make a VAX FORTRAN routine add a NUL character to a CHARACTER string before calling a VAX C function. You may also write code that explicitly gets the length of a VAX FORTRAN string from its descriptor and carefully pads the string with spaces after modifying it. An example later in this section shows a C function that carefully produces a proper string for VAX FORTRAN.

Example 13–5 shows a VAX C function calling a VAX FORTRAN subprogram with a variety of data types. For most scalar types, VAX FORTRAN expects arguments to be passed by reference but character data is passed by descriptor.

Example 13–5: VAX C Function Calling a VAX FORTRAN Subprogram

```
/*
 * Beginning of VAX C function:
 */

#include <stdio.h>
#include <descrip.h>          /* Get layout of descriptors */
extern int fort();          /* Declare FORTRAN function */

main()
{
    int i = 508;
    float f = 649.0;
    double d = 91.50;
    struct {
        short s;
        float f;
    } s = {-2, -3.14};
    auto $DESCRIPTOR(string1, "Hello, FORTRAN");
    struct dsc$descriptor_s string2;

    /* "string1" is a FORTRAN-style string declared and initialized using the
     * $DESCRIPTOR macro. "string2" is also a FORTRAN-style string, but we are
     * declaring and initializing it by hand. */
    string2.dsc$b_dtype = DSC$K_DTYPE_T; /* type is CHARACTER */
    string2.dsc$b_class = DSC$K_CLASS_S; /* string descriptor */
    string2.dsc$w_length = 3; /* three characters in string */
    string2.dsc$a_pointer = "bye"; /* pointer to string value */

    printf("FORTRAN result is %d\n", fort(&i, &f, &d, &s, &string1, &string2));
} /* End of VAX C function */
```

(continued on next page)

Example 13-5 (Cont.): VAX C Function Calling a VAX FORTRAN Subprogram

```
C
C   Beginning of VAX FORTRAN subprogram:
C
INTEGER FUNCTION FORT(I, F, D, S, STRING1, STRING2)
INTEGER I
REAL F
DOUBLE PRECISION D
STRUCTURE /STRUCT/
INTEGER*2 SHORT
REAL FLOAT
END STRUCTURE
RECORD /STRUCT/ S
C   You can tell FORTRAN to use the length in the descriptor
C   as done here for STRING1, or you can tell FORTRAN to ignore the
C   descriptor and assume the string has a particular length as done
C   for STRING2. This choice is up to you.
CHARACTER*(*) STRING1
CHARACTER*3 STRING2

WRITE(5, 10) I, F, D, S.SHORT, S.FLOAT, STRING1, STRING2
10  FORMAT(1X, I3, F8.1, D10.2, I7, F10.3, 1X, A, 2X, A)
FORT = -15
RETURN
END
C   End of VAX FORTRAN subprogram
```

Output from Example 13-5 is as follows:

```
508  649.0 0.92D+02      -2      -3.140 Hello, FORTRAN  bye
FORTRAN result is -15
```

Example 13-6 shows a VAX FORTRAN subprogram calling a VAX C function. Since the VAX C function is called from FORTRAN as a subroutine and not as a function, the VAX C function is declared to have a return value of **void**.

Example 13–6: VAX FORTRAN Subprogram Calling a VAX C Function

```
C
C   Beginning of VAX FORTRAN subprogram:
C
      INTEGER I
      REAL F(3)
      CHARACTER*10 STRING

C   Since this program does not have a C main program and you want
C   to use VAX C RTL functions from the C subroutine, you must call
C   VAXC$CRTL_INIT to initialize the VAX C RTL.
      CALL VAXC$CRTL_INIT

      I = -617
      F(1) = 3.1
      F(2) = 0.04
      F(3) = 0.0016
      STRING = 'HELLO'

      CALL CSUBR(I, F, STRING)
      END
C   End of VAX FORTRAN subprogram

/*
 * Beginning of VAX C function:
 */
#include <stdio.h>
#include <descrip.h>                /* Get layout of descriptors */

void csubr(i, f, string)
    int *i;                        /* FORTRAN integer, by reference */
    float f[3];                    /* FORTRAN array, by reference */
    struct dsc$descriptor_s *string; /* FORTRAN character, by descriptor */
{
    int j;

    printf("i = %d\n", *i);
    for (j = 0; j < 3; ++j)
        printf("f[%d] = %f\n", j, f[j]);

    /* Since FORTRAN character data is not NUL-terminated, you must use
     * a counted loop to print the string.
     */
    printf("string = \");
    for (j = 0; j < string->dsc$w_length; ++j)
        putchar(string->dsc$a_pointer[j]);
    printf("\n");
} /* End of VAX C function */
```

Output from Example 13-6 is as follows:

```
i = -617
f[0] = 3.100000
f[1] = 0.040000
f[2] = 0.001600
string = "HELLO      "
```

Example 13-7 shows a C function that acts like a CHARACTER*(*) function in VAX FORTRAN.

Example 13-7: VAX C Function Emulating a VAX FORTRAN CHARACTER*(*) Function

```
C
C      Beginning of VAX FORTRAN program:
C
C      CHARACTER*9 STARS, C
C
C      Call a C function to produce a string of three "*" left-justified
C      in a nine-character field.
C      C = STARS(3)
C
C      WRITE(5, 10) C
10     FORMAT(1X, "'", A, "'")
C      END
C      End of VAX FORTRAN program
C
/*
*      Beginning of VAX C function:
*/
#include <descrip.h>                                /* Get layout of descriptors */
/* Routine "stars" is equivalent to a FORTRAN function declared as
* follows:
*
*      CHARACTER*(*) FUNCTION STARS(NUM)
*      INTEGER NUM
*
* Note that a FORTRAN CHARACTER function has an extra entry added to
* the argument list to represent the return value of the CHARACTER
* function. This entry, which appears first in the argument list,
* is the address of a completely filled-in character descriptor. Since
* the C version of a FORTRAN character function explicitly uses this
* extra argument list entry, the C version of the function is void!
*
* This example function returns a string that contains the specified
* number of asterisks (or "stars").
*/
```

(continued on next page)

Example 13-7 (Cont.): VAX C Function Emulating a VAX FORTRAN CHARACTER*(*) Function

```
void stars(return_value, num_stars)
    struct dsc$descriptor_s *return_value;
                                /* FORTRAN return value */
    int *num_stars;              /* Number of "stars" to create */
{
    int i, limit;

    /* A FORTRAN string is truncated if it is too large for the memory area
     * allocated, and it is padded with spaces if it is too short. Set limit
     * to the number of stars to put in the string given the size of the area
     * used to store it. */
    if (*num_stars < return_value->dsc$w_length)
        limit = *num_stars;
    else
        limit = return_value->dsc$w_length;

    /* Create a string of stars of the specified length up to the limit of the
     * string size. */
    for (i = 0; i < limit; ++i)
        return_value->dsc$a_pointer[i] = '*';

    /* Pad rest of string with spaces, if necessary. */
    for (; i < return_value->dsc$w_length; ++i)
        return_value->dsc$a_pointer[i] = ' ';
} /* End of VAX C Function */
```

Output from Example 13-7 is as follows:

```
****      "
```

13.3.2 Calling VAX MACRO

You can easily call a VAX MACRO routine from VAX C, or vice versa. However, like all interlanguage calls, it is necessary to take care in making sure that the actual arguments correspond to the expected formal parameter types. Also, it is necessary to remember that C strings are NUL-terminated and to take special action in either the MACRO routine or the C routine to allow for this.

Example 13-8 shows a macro routine that calls a C routine with three arguments, passing one by value, one by reference, and one by descriptor. It is followed by the source for the called C routine.

Example 13-8: VAX MACRO Program Calling a VAX C Function

```
-----  
; Beginning of MACRO program  
-----  
    .extrn  dbroutine          ; The C routine  
-----  
; Local Data  
-----  
    .psect      data          rd,nowrt,noexe  
ft$$t_part_num:      .ascii /WidgitGadget/  
ft$$t_query_mode:    .ascii /I/  
ft$$s_query_mode =   <. - ft$$t_query_mode>  
ft$$l_protocol_buff: .blk1      1  
ft$$kd_part_num_dsc:  
                    .word  12  
                    .word   0  
                    .address ft$$t_part_num  
-----  
; Entry Point  
-----  
    .psect      ft_code       rd,nowrt,exe  
    .entry dbtest             ^m<r2,r3,r4,r5,r6,r7,r8>  
  
;+  
; call C routine for data base lookup  
;-  
    movl        #1,r3  
    pushal     ft$$kd_part_num_dsc      ; Descriptor for part number  
    pushal     ft$$t_query_mode        ; Mode to call  
    pushl      #1                       ; status  
    calls      #3, dbroutine            ; Check the data base  
99$:  
    ret  
  
    .end dbtest  
-----  
; End of MACRO program  
-----  
/*  
 * Beginning of VAX C code for dbroutine:  
 */  
  
#include <stdio.h>  
#include <descrip.h>
```

(continued on next page)

Example 13–8 (Cont.): VAX MACRO Program Calling a VAX C Function

```
/* Structure pn_desc is the format of the descriptor
   passed by the macro routine. */
extern struct
    mydescriptor
    {
        short pn_len;
        short pn_zero;
        char *pn_addr;
    };

int dbroutine (status, action, name_dsc )
int status; /* Passed by value */
char *action; /* Passed by reference */
struct mydescriptor *name_dsc; /* Passed by descriptor */
{
    char *part_name;

    /* Allocate space to put the null-padded name string. */
    part_name = malloc(name_dsc->pn_len + 1);
    memcpy( part_name,name_dsc -> pn_addr ,name_dsc -> pn_len);

    /* Remember that C array bounds start at 0 */
    part_name[name_dsc -> pn_len] = '\0';

    printf (" Status is %d\n", status);
    printf (" Length is %d\n",name_dsc -> pn_len);
    printf (" Part_name is %s\n",part_name);
    printf (" Request is %c\n",*action);
    status = 1;
    return(status);
} /* End of VAX C code for dbroutine */
```

Output from Example 13–8 is as follows:

```
Status is 1
Length is 12
Part_name is WidgitGadget
Request is I
```

Example 13–9 shows a VAX C program that calls a VAX MACRO program.

Example 13-9: VAX C Program Calling a VAX MACRO Program

```
/*
 * Beginning of VAX C function
 */

#include <stdio.h>
#include <descrip.h>

int zapit( int status, int *action, struct dsc$descriptor_s *descript);

main()
{
    int status=255, argh = 99;
    int *action = &argh;
    $DESCRIPTOR(name_dsc,"SuperEconomySize");

    printf(" Before calling ZAPIT: \n");
    printf(" Status was %d \n",status);
    printf(" Action contained %d and *action contained %d \n" ,action, *action);
    printf(" And the thing described by the descriptor was %s\n",
           name_dsc.dsc$a_pointer);

    if (zapit(status,action,&name_dsc) && 1)
    {
        printf(" Ack, the world has been zapped! \n");
        printf(" Status is %d \n",status);
        printf(" Action contains %d and *action contains %d \n" ,action, *action);
        printf(" And the address of the thing described by the descriptor is %d\n",
               name_dsc.dsc$a_pointer);
    }
} /* End of VAX C function

;-----
; Beginning of VAX MACRO source code for zapit
;-----
; Entry Point
;-----
        .psect      ft_code      rd,nowrt,exe
        .entry zapit      ^m<r2,r3,r4,r5,r6,r7,r8>

;+
; Maliciously change parameters passed by the C routine.
;
; The first parameter is passed by value, the second by
; reference, and the third by descriptor.
;-

        movl      4(ap), @8(ap)      ;Change the by-reference parameter
                                   ;to the first parameter's value.

        movl      12(ap), r2
        movl      #0,4(r2)          ;Zap address of string in descriptor
```

(continued on next page)

Example 13–9 (Cont.): VAX C Program Calling a VAX MACRO Program

```
    ; Return -1 to signal successful destruction
    movl    #-1,r0
    ret
    .end
;-----
; End of VAX MACRO source code for zapit
;-----
```

Output from Example 13–9 is as follows:

```
Before calling ZAPIT:
Status was 255
Action contained 2146269556 and *action contained 99
And the thing described by the descriptor was SuperEconomySize
Ack, the world has been zapped!
Status is 255
Action contains 2146269556 and *action contains 255
And the address of the thing described by the descriptor is 0
```

13.3.3 Calling VAX BASIC

Calling routines written in VAX BASIC from VAX C programs, or vice versa, is straightforward. By default, VAX BASIC passes arguments by reference, except for arrays and strings, which are passed by descriptor. In some cases, these defaults may be overridden by explicitly specifying the desired parameter-passing mechanisms in the VAX BASIC program. However, if an argument is a constant or an expression, the actual argument passed refers to a local copy of the specified argument's value.

Strings in VAX BASIC are not terminated by a NUL character, which is done by VAX C. As a result, passing strings between VAX BASIC and VAX C routines requires you to do additional work. You may choose to add an explicit NUL character to a VAX BASIC string before passing it to a VAX C routine, or you may prefer to code the VAX C routine to obtain the string's length from its descriptor.

Example 13–10 shows a VAX C program that calls a VAX BASIC function with a variety of argument data types.

Example 13-10: VAX C Function Calling a VAX BASIC Function

```
/*
 * Beginning of VAX C function:
 */

#include <stdio.h>
#include <descrip.h>

extern      int      basfunc ();

main ()
{
    int      i = 508;
    float    f = 649.0;
    double   d = 91.50;
    struct
    {
        short    s;
        float    f;
    }
    s = { -2, -3.14 };
    $DESCRIPTOR (string1, "A C string");

    printf ("BASIC returned %d\n",
            basfunc (&i, &f, &d, &s, &string1, "bye"));
} /* End of VAX C function */

REM Beginning of the VAX BASIC program
FUNCTION INTEGER basfunc (INTEGER i, REAL f, DOUBLE d, x s, &
                        STRING string1, &
                        STRING string2 = 3 BY REF)

RECORD      x
    WORD     s
    REAL     f
END RECORD x

PRINT 'i = ' ; i
PRINT 'f = ' ; f
PRINT 'd = ' ; d
PRINT 's::s = ' ; s::s
PRINT 's::f = ' ; s::f
PRINT 'string1 = ' ; string1
PRINT 'string2 = ' ; string2

END FUNCTION -15
REM End of the VAX BASIC program
```

Output from Example 13-10 is as follows:

```
i = 508
f = 649
d = 91.5
s::s = -2
s::f = -3.14
string1 = A C string
string2 = bye
BASIC returned -15
```

Example 13–11 shows a VAX BASIC program that calls a VAX C function.

Example 13–11: VAX BASIC Program Calling a VAX C Function

```
REM Beginning of the VAX BASIC program:
PROGRAM example

    EXTERNAL STRING FUNCTION cfunc (INTEGER BY VALUE, &
                                    INTEGER BY VALUE, &
                                    STRING BY DESC)

    s$ = cfunc (5, 3, "abcdefghi")
    PRINT "substring is "; s$

END PROGRAM
REM End of the VAX BASIC program

/*
 * Beginning of VAX C function:
 */
#include <descrip.h>

/*
 * This routine simulates a VAX BASIC function whose return
 * value is a STRING. It returns the substring that is 'length'
 * characters long, starting from the offset 'offset' (0-based)
 * in the input string described by the descriptor pointed to
 * by 'in_str'.
 */
void cfunc (struct dsc$descriptor_s *out_str,
            int offset,
            int length,
            struct dsc$descriptor_s *in_str)
{
    /* Declare a string descriptor for the substring. */
    struct dsc$descriptor temp;

    /* Check that the desired substring is wholly
       within the input string. */
    if (offset + length > in_str->dsc$w_length)
        return;

    /* Fill in the substring descriptor. */
    temp.dsc$w_length = length;
    temp.dsc$a_pointer = in_str->dsc$a_pointer + offset;
    temp.dsc$b_dtype = DSC$K_DTYPE_T;
    temp.dsc$b_class = DSC$K_CLASS_S;

    /* Copy the substring to the return string */
    str$copy_dx (out_str, &temp);
} /* End of VAX C function */
```

Output from Example 13–11 is as follows:

```
substring is fgh
```

13.3.4 Calling VAX Pascal

Like VAX FORTRAN and VAX BASIC, there are certain considerations that you must take into account when calling VAX Pascal from VAX C and vice versa. When calling VAX Pascal from VAX C, VAX Pascal expects all parameters to be passed by reference. In VAX Pascal, there are two different types of semantics: value and variable. The value semantics in VAX Pascal are different from passing by value in VAX C. Because they are different, you must specify the address of the C parameter.

VAX Pascal also expects all strings to be passed by descriptor. If you use the CLASS_S descriptor, the string is passed by using VAX Pascal semantics. If the content of the string is changed, it is not reflected back to the caller.

Example 13-12 is an example of how to call a VAX Pascal routine from VAX C.

Example 13-12: VAX C Function Calling a VAX Pascal Routine

```
/*
 * Beginning of VAX C function:
 */

#include descrip

/* This program demonstrates how to call a Pascal routine
   from a C function. */

/* A Pascal routine called by a C function */
extern      void      Pascal_Routine ();

main ()
{
    struct dsc$descriptor_s to_Pascal_by_desc;
    char *Message = "The_Max_Num";
    int to_Pascal_by_value = 100,
        to_Pascal_by_ref = 50;

    /* Construct the descriptor. */
    to_Pascal_by_desc.dsc$a_pointer = Message;
    to_Pascal_by_desc.dsc$w_length = strlen (Message);
    to_Pascal_by_desc.dsc$b_class = DSC$K_CLASS_S;
    to_Pascal_by_desc.dsc$b_dtype = DSC$K_DTYPE_T;

    /* Pascal expects a calling routine to pass parameters by reference. */
    Pascal_Routine(&to_Pascal_by_value, &to_Pascal_by_ref, &to_Pascal_by_desc);
}
```

(continued on next page)

Example 13-12 (Cont.): VAX C Function Calling a VAX Pascal Routine

```
printf ("\nWhen returned from Pascal:\nto_Pascal_by_value is still \
%d\nBut to_Pascal_by_ref is %d\nand Message is still %s\n",
        to_Pascal_by_value, to_Pascal_by_ref,
        to_Pascal_by_desc.dsc$a_pointer);
} /* End of VAX C function */
{
  Beginning of VAX Pascal routine
}

MODULE C_PASCAL(OUTPUT);

{ This Pascal routine calls the Pascal MAX function
  to determine the maximum value between
  'from_c_by_value' and 'from_c_by_ref', and then
  assigns the result back to 'from_c_by_ref'.
  It also tries to demonstrate the results of passing
  a by-descriptor mechanism.
  It is called from a C main function.
}
[GLOBAL]PROCEDURE Pascal_Routine
(
    from_c_by_value :INTEGER;
  VAR from_c_by_ref :INTEGER;
    from_c_by_desc  :[ CLASS_S ] PACKED ARRAY [1..11:INTEGER] OF CHAR
);

VAR
  today_is : PACKED ARRAY [1..11] OF CHAR;

BEGIN

  { Display the contents of formal parameters. }
  Writeln;
  Writeln ('Parameters passed from C function: ');
  Writeln ('from_c_by_value: ', from_c_by_value:4);
  Writeln ('from_c_by_ref: ', from_c_by_ref:4);
  Writeln ('from_c_by_desc: ', from_c_by_desc);

  { Assign the maximum value into 'from_c_by_ref' }
  from_c_by_ref := MAX (from_c_by_value, from_c_by_ref);

  { Change the content of 'from_Pascal_by_value' --
    to show that the value did not get
    reflected back to the caller.
  }
  from_c_by_value := 20;
```

(continued on next page)

Example 13-12 (Cont.): VAX C Function Calling a VAX Pascal Routine

```
{ Put the results of DATE into 'from_c_by_desc`
  to show that the CLASS_S is only valid with
  comformant strings passed by value.
}
DATE (today_is);
from_c_by_desc := today_is;
WRITELN ('*****');
WRITELN ('from_c_by_desc is changed to today''s date: "',
        from_c_by_desc, "'");
WRITELN ('BUT, this will not reflect back to the caller.');
```

END;
END.
{
 End of VAX Pascal routine
}

Output from Example 13-12 is as follows:

```
from_c_by_value: 100
from_c_by_ref: 50
from_c_by_desc: The_Max_Num
*****
from_c_by_desc is changed to today's date "26-MAY-1988"
BUT, this will not reflect back to the caller.
```

```
When returned from Pascal:
to_Pascal_by_value is still 100
to_Pascal_by_ref is 100
and Message is still The_Max_Num
```

There are also some considerations when calling VAX C from VAX Pascal. For example, you can use mechanism specifiers such as `%IMMED`, `%REF`, and `%STDESCR` in VAX Pascal. When you use the `%IMMED` mechanism specifier, the compiler passes a copy of a value rather than an address. When you use the `%REF` mechanism specifier, the address of the actual parameter is passed to the called routine, which is then allowed to change the value of the corresponding actual parameter. When you use the `%STDESCR` mechanism specifier, the compiler generates a fixed-length descriptor of a character-string variable and passes its address to the called routine. For more information on these mechanism specifiers and others, see the VAX Pascal documentation.

Another consideration is that VAX Pascal does not NUL pad strings. Therefore, you must add a NUL character to make the string a VAX C string. Also, when passing a string from VAX Pascal to VAX C, you can declare a structure declaration in VAX C that corresponds to the VAX Pascal `VARYING TYPE` declaration.

Example 13-13 shows an example of how to call VAX C from VAX Pascal.

Example 13-13: VAX Pascal Program Calling a VAX C Function

```
{
  Beginning of VAX Pascal function:
}

PROGRAM PASCAL_C (OUTPUT);

CONST
  STRING_LENGTH = 80;

TYPE
  STRING = VARYING [STRING_LENGTH] OF CHAR;

VAR
  by_value : INTEGER;
  by_ref   : STRING;
  by_desc  : PACKED ARRAY [1..10] OF CHAR;

[EXTERNAL]
PROCEDURE VAXC$CRTL_INIT; EXTERN;

[EXTERNAL]
PROCEDURE c_function
(  %immed      to_c_by_value : INTEGER;
  %ref        to_c_by_ref   : STRING ;
  %stdescr    to_c_by_desc  : PACKED ARRAY [1..10] OF CHAR
); EXTERN;

BEGIN
  { Establish appropriate VAX C RTL environment for
    calling VAX C RTL from Pascal.
  }
  VAXC$CRTL_INIT;

  by_value := 1;

  {
    NOTE
    Pascal does not NUL pad a string.
    Therefore, the LENGTH built-in function counts
    the NUL pad character while the C RTL strlen function
    does not include the terminating NUL character.
  }

  by_ref := 'TO_C_BY_REF'(0)'';
  by_desc := 'TERM'(0)'';

  { Call a C function by passing parameters
    using foreign semantics.
  }
  c_function (by_value, by_ref, by_desc);
```

(continued on next page)

Example 13-13 (Cont.): VAX Pascal Program Calling a VAX C Function

```
WRITELN;
WRITELN;
WRITELN ('*****');
WRITELN ('After calling C_FUNCTION: ');
WRITELN;
WRITELN ('to_c_by_value is still ',by_value:3);
WRITELN ('however, to_c_by_ref contains ',by_ref,
        ' (aka Your Terminal Type)');
WRITELN ('and, to_c_by_desc still contains ',by_desc);

END.
{
  End of VAX Pascal program
}

/*
 * Beginning of VAX C function:
 *
 * A C function called from the Pascal routine.
 * The parameters are passed to a C function
 * by value, by reference, and by descriptor,
 * respectively.
 */
#include descrip

/* A Pascal style of VARYING data type. */
struct Pascal_VARYING
{
  unsigned short    length;
  char              string[80];
};

/* This C function calls the C RTL function getenv() and puts
 * your terminal type in 'from_Pascal_by_ref'.
 * It is called from a Pascal program.
 */
void      c_function (unsigned char      from_Pascal_by_value,
                    struct Pascal_VARYING *from_Pascal_by_ref,
                    struct dsc$descriptor_s *from_Pascal_by_desc
                    )
{
  char *term;

  /* Display the contents of formal parameters. */
  printf ("\nParameters passed from Pascal:\n");
  printf ("from_Pascal_by_value: %d\nfrom_Pascal_by_ref: %s\n"
from_Pascal_by_desc: %s\n", from_Pascal_by_value,
                    from_Pascal_by_ref -> string,
                    from_Pascal_by_desc -> dsc$a_pointer);
}
```

(continued on next page)

Example 13–13 (Cont.): VAX Pascal Program Calling a VAX C Function

```
if ((term = getenv(from_Pascal_by_desc -> dsc$a_pointer)) != 0)
{
    /* Fill 'from_Pascal_by_ref' with new value. */
    strcpy (from_Pascal_by_ref -> string, term);
    from_Pascal_by_ref -> length = strlen (term);

    /* Change the contents of 'from_Pascal_by_value' --
    * to demonstrate that the value did not get
    * reflected back to the calling routine.
    */
    from_Pascal_by_value = from_Pascal_by_desc -> dsc$w_length
        + from_Pascal_by_ref -> length;
}
else
    printf ("\ngetenv\(\"TERM\") is undefined.");
} /* End of VAX C function */
```

Output from Example 13–13 is as follows:

```
Parameters passed from Pascal:
from_Pascal_by_value: 1
from_Pascal_by_ref: TO_C_BY_REF
from_Pascal_by_desc: TERM

*****
After calling C_FUNCTION:

to_c_by_value is still 1
however, to_c_by_ref contains vt200-80 (aka Your Terminal Type)
and, to_c_by_desc still contains TERM
```

13.4 Sharing Global Data

The following sections describe the methods involved in sharing VAX C program sections with data declared in other VAX-native languages.

13.4.1 Sharing Program Sections with FORTRAN Common Blocks

In a FORTRAN program, separately compiled procedures can share data in declared common blocks, which specify the names of one or more variables to be placed in them. Each named common block represents a separate program section. Each procedure that declares the common block with the same name can access the same variable.

Example 13–14 shows a VAX C **extern** variable that corresponds to a FORTRAN common block with the same name.

The FORTRAN program PRSTRING.FOR contains the following lines of code:

```
SUBROUTINE PRSTRING
CHARACTER*20 STRING
COMMON /XYZ/ STRING

TYPE 20, STRING
20 FORMAT (' ',A20)
RETURN
END
```

Example 13–14: Sharing Data with a FORTRAN Program in Named Program Sections

```
/* VAX C program STRING.C contains the following lines of      *
 * code:                                                         */
main()
{
    extern char xyz[20];
    strncpy(xyz,"This is a string      ", sizeof xyz);
    prstring();
}
```

In Example 13–14, the VAX C **extern** variable `xyz` corresponds to the FORTRAN common block named `XYZ`. The FORTRAN procedure displays the data in the block. When sharing program sections, both programs should declare corresponding variables to be of the same type.

To share data in more than one variable in a program section with a FORTRAN program, the VAX C variables must be declared within a structure, as shown in Example 13–15.

Example 13–15: Sharing Data with a FORTRAN Program in a VAX C Structure

```
/* VAX C program NUMBERS.C contains the following lines of  *
 * code:                                                    */
struct xs
{
    int first;
    int second;
    int third;
};

main()
{
    extern struct xs numbers;

    numbers.first = 1;
    numbers.second = 2;
    numbers.third = 3;
    fnum();
}
```

The FORTRAN program FNUM.FOR contains the following lines of code:

```
SUBROUTINE FNUM
INTEGER*4 INUM, JNUM, KNUM
COMMON /NUMBERS/ INUM, JNUM, KNUM

TYPE 10, (INUM, JNUM, KNUM)
10 FORMAT (3I8)
RETURN
END
```

In Example 13–15, the **int** variables declared in the VAX C structure `numbers` correspond to the FORTRAN `INTEGER*4` variables in the `COMMON` of the same name.

13.4.2 Sharing Program Sections with PL/I Externals

A VAX PL/I variable with the `EXTERNAL` attribute corresponds to a FORTRAN common block and to a VAX C **extern** variable. Example 13–16 and Example 13–17 show how a program section is shared between VAX C and VAX PL/I.

A PL/I `EXTERNAL CHARACTER` attribute corresponds to a VAX C **extern char** variable, but PL/I character strings are not necessarily NUL-terminated. In Example 13–16, VAX C and VAX PL/I use the same variable to manipulate the character string that resides in a program section named `XYZ`.

Example 13–16: Sharing Data with a PL/I Program in Named Program Sections

```
/* VAX C program STRING.C contains the following lines of      *
 * code:                                                         */
main()
{
    extern char xyz[20];

    strncpy(xyz,"This is a string    ", sizeof xyz);
    prstring();
}
```

The PL/I program PRSTRING.PLI contains the following lines of code:

```
PRSTRING: PROCEDURE;

    DECLARE XYZ EXTERNAL CHARACTER(20);

    PUT SKIP LIST(XYZ);
    RETURN;

END PRSTRING;
```

The PL/I procedure PRSTRING writes out the contents of the external variable XYZ.

PL/I also has a structure type similar (in its internal representation) to the **struct** keyword in VAX C. Moreover, VAX PL/I can output aggregates, such as structures and arrays, in fairly simple stream-output statements; consider Example 13–17.

The PL/I program FNUM.PLI contains the following lines of code:

```
FNUM: PROCEDURE;
    /* EXTERNAL STRUCTURE CONTAINING THREE INTEGERS */
    DECLARE 1 NUMBERS EXTERNAL,
            2 FIRST FIXED(31),
            2 SECOND FIXED(31),
            2 THIRD FIXED(31);

    PUT SKIP LIST('Contents of structure:',NUMBERS);
    RETURN;
END FNUM;
```

The PL/I procedure FNUM writes out the complete contents of the external structure NUMBERS; the structure members are written out in the order of their storage in memory, which is the same as for a VAX C structure.

Example 13–17: Sharing Data with a PL/I Program in a VAX C Structure

```
/* VAX C program NUMBERS.C contains the following lines of   *
 * code:                                                         */
struct xs
{
    int first;
    int second;
    int third;
};
main()
{
    extern struct xs numbers;

    numbers.first = 1;
    numbers.second = 2;
    numbers.third = 3;
    fnum();
}
```

13.4.3 Sharing Program Sections with MACRO Programs

In a MACRO program, the `.PSECT` directive sets up a separate program section that can store data or MACRO instructions. The attributes in the `.PSECT` directive describe the contents of the program section.

Example 13–18 shows how to set up a `psect` in a MACRO program that allows data to be shared with a VAX C program.

The MACRO source code file `SET_VALUE.MAR` is as follows:

```
.entry set_value, ^M<>

movl    #1, first
movl    #2, second
movl    #3, third
ret

.psect  example pic,usr,ovr,rel,dbl,shr,-
        noexe,rd,wrt,novec,long

first:  .blk1
second: .blk1
third:  .blk1

.end
```

Example 13–18: Sharing Data with a MACRO Program in a VAX C Structure

```
/* VAX C program NUMBERS.C contains the following lines of  *
 * code:                                                    */
struct xs
{
    int first;
    int second;
    int third;
} example;

main()
{
    set_value();

    printf("example.first = %d\n", example.first);
    printf("example.second = %d\n", example.second);
    printf("example.third = %d\n", example.third);
}
```

The MACRO program initializes the locations first, second, and third in the psect named example and passes these values to the VAX C program. The locations are referenced in the VAX C program as members of the external structure named example.

13.5 VMS Run-Time Library Routines

The VMS RTL is a library of prewritten, commonly-used routines that perform a wide variety of functions. These routines are grouped according to the types of tasks they perform, and each group has a prefix that identifies those routines as members of a particular VMS RTL facility. Table 13–8 lists all the language-independent, run-time library facility prefixes and the types of tasks each facility performs.

Table 13–8: Run-Time Library Facilities

Facility Prefix	Types of Tasks Performed
DTK\$	DECTalk routines that are used to control the DIGITAL DECTalk device.
LIB\$	Library routines that obtain records from devices, manipulate strings, convert data types for I/O, allocate resources, obtain system information, signal exceptions, establish condition handlers, enable detection of hardware exceptions, and process cross-reference data.
MTH\$	Mathematics routines that perform arithmetic, algebraic, and trigonometric calculations.
OTS\$	General-purpose routines that perform tasks such as data type conversions as part of a compiler's generated code.
SMG\$	Screen management routines that are used in designing, composing, and keeping track of complex images on a video screen.
STR\$	String manipulation routines that perform such tasks as searching for substrings, concatenating strings, and prefixing and appending strings.

13.6 VMS System Services Routines

System services are prewritten system routines that perform a variety of tasks, such as controlling processes, communicating among processes, and coordinating I/O.

Unlike the VMS RTL routines, which are divided into groups by facility, all system services share the same facility prefix (SYS\$). However, these services are logically divided into groups that perform similar tasks. Table 13–9 describes these groups.

Table 13–9: System Services

Group	Types of Tasks Performed
AST	Allows processes to control the handling of ASTs.
Change Mode	Changes the access mode of particular routines.
Condition Handling	Designates condition handlers for special purposes.
Event Flag	Clears, sets, reads, and waits for event flags, and associates with event flag clusters.
Information	Returns information about the system, queues, jobs, processes, locks, and devices.
Input/Output	Performs I/O directly, without going through VAX RMS.
Lock Management	Enables processes to coordinate access to shareable system resources.
Logical Names	Provides methods of accessing and maintaining pairs of character-string logical names and equivalence names.
Memory Management	Increases or decreases available virtual memory, controls paging and swapping, and creates and accesses shareable files of code or data.
Process Control	Creates, deletes, and controls execution of processes.
Security	Enhances the security of VMS systems.
Time and Timing	Schedules events, and obtains and formats binary time values.

13.7 Calling Routines

The basic steps for calling routines are the same whether you are calling a routine written in VAX C, a routine written in some other VAX language, a system service, or a VMS RTL routine. The following sections outline the procedures for calling non-VAX C routines.

13.7.1 Determining the Type of Call

Before calling an external routine, you must first determine whether the call should be a procedure call or a function call. Call a routine as a procedure if it does not return a value. Call a routine as a function if it returns any type of value.

13.7.2 Declaring an External Routine and Its Arguments

To call an external routine or system routine, you need to declare it as an external function and to declare the names, data types, and passing mechanisms of its arguments. Arguments can be either required or optional.

Include the following information in a routine declaration:

- The name of the external routine
- The data types of all the routine parameters (optional)
- The data type of the return value if it is a function
- The **void** keyword if it is a procedure

The following example shows how to declare an external routine and its arguments:

```
char func_name (int x, char y);
```

13.7.3 Calling the External Routine

After declaring an external routine, you can invoke it. To invoke a function, you must specify the name of the routine being invoked and all arguments required for that routine. Make sure the data types for the actual arguments you are passing coincide with those of the parameters you declared earlier, and with those declared in the routine. The following example shows how to invoke the function declared in Section 13.7.2:

```
ret_status = func_name(1, 'a');
```

If you do not want to specify a value for a required parameter, pass a null argument by inserting a 0 as a placeholder in the argument list.

13.7.4 System Routine Arguments

All system routine arguments are described in terms of the following information:

- VMS usage
- Data type
- Type of access allowed
- Passing mechanism

VMS usages are data structures that are layered on the standard VMS data types. For example, the VMS usage `mask_longword` signifies an unsigned longword integer that is used as a bit mask, and the VMS usage `floating_point` represents any VMS floating-point data type. Table 13–10 lists all the VMS usages and the VAX C types you need to implement them.

Table 13–10: VAX C Implementation

VMS Data Type	VAX C Declaration
<code>access_bit_names</code>	user-defined ¹
<code>access_mode</code>	unsigned char
<code>address</code>	int *pointer ^{2,4}
<code>address_range</code>	int *array [2] ^{2,3,4}
<code>arg_list</code>	user-defined ¹
<code>ast_procedure</code>	pointer to a function ²
<code>boolean</code>	unsigned long int
<code>byte_signed</code>	char
<code>byte_unsigned</code>	unsigned char
<code>channel</code>	unsigned short int
<code>char_string</code>	char array[n] ^{3,5}
<code>complex_number</code>	user-defined ¹
<code>cond_value</code>	unsigned long int
<code>context</code>	unsigned long int
<code>date_time</code>	user-defined ¹
<code>device_name</code>	char array[n] ^{3,5}
<code>ef_cluster_name</code>	char array[n] ^{3,5}
<code>ef_number</code>	unsigned long int
<code>exit_handler_block</code>	user-defined ¹

¹The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is more suitable to specific applications.

²The term `pointer` refers to several declarations involving pointers. Pointers are declared with special syntax and are associated with the data type of the object being pointed to. This object is often user-defined.

³The term `array` denotes the syntax of a VAX C array declaration.

⁴The data type specified can be changed to any valid VAX C data type.

⁵The size of the array must be substituted for `n`.

(continued on next page)

Table 13–10 (Cont.): VAX C Implementation

VMS Data Type	VAX C Declaration
fab	struct FAB #include fab from text library
file_protection	unsigned short int, or user-defined ¹
floating_point	float or double
function_code	unsigned long int or user-defined ¹
identifier	int *pointer ^{2,4}
io_status_block	user-defined ¹
item_list_2	user-defined ¹
item_list_3	user-defined ¹
item_list_pair	user-defined ¹
item_quota_list	user-defined ¹
lock_id	unsigned long int
lock_status_block	user-defined ¹
lock_value_block	user-defined ¹
logical_name	char array[n] ^{3,5}
longword_signed	long int
longword_unsigned	unsigned long int
mask_byte	unsigned char
mask_longword	unsigned long int
mask_quadword	user-defined ¹
mask_word	unsigned short int
null_arg	unsigned long int
octaword_signed	user-defined ¹
octaword_unsigned	user-defined ¹

¹The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is more suitable to specific applications.

²The term pointer refers to several declarations involving pointers. Pointers are declared with special syntax and are associated with the data type of the object being pointed to. This object is often user-defined.

³The term array denotes the syntax of a VAX C array declaration.

⁴The data type specified can be changed to any valid VAX C data type.

⁵The size of the array must be substituted for n.

(continued on next page)

Table 13–10 (Cont.): VAX C Implementation

VMS Data Type	VAX C Declaration
page_protection	unsigned long int
procedure	pointer to function ²
process_id	unsigned long int
process_name	char array[n] ^{3,5}
quadword_signed	user-defined ¹
quadword_unsigned	user-defined ¹
rights_holder	user-defined ¹
rights_id	unsigned long int
rab	#include rab struct RAB
section_id	user-defined ¹
section_name	char array[n] ^{3,5}
system_access_id	user-defined ¹
time_name	char array[n] ^{3,5}
uic	unsigned long int
user_arg	user-defined ¹
varying_arg	user-defined ¹
vector_byte_signed	char array[n] ^{3,5}
vector_byte_unsigned	unsigned char array[n] ^{3,5}
vector_longword_signed	long int array[n] ^{3,5}
vector_longword_unsigned	unsigned long int array[n] ^{3,5}
vector_quadword_signed	user-defined ¹
vector_quadword_unsigned	user-defined ¹
vector_word_signed	short int array[n] ^{3,5}
vector_word_unsigned	unsigned short int array[n] ^{3,5}

¹The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is more suitable to specific applications.

²The term pointer refers to several declarations involving pointers. Pointers are declared with special syntax and are associated with the data type of the object being pointed to. This object is often user-defined.

³The term array denotes the syntax of a VAX C array declaration.

⁵The size of the array must be substituted for n.

(continued on next page)

Table 13–10 (Cont.): VAX C Implementation

VMS Data Type	VAX C Declaration
word_signed	short int
word_unsigned	unsigned short int

If a system routine argument is optional, it will be indicated in the format section of the routine description in one of two ways, as follows:

- [,optional-argument]
- ,[optional-argument]

If the comma appears outside the brackets (,[optional-argument]), you must pass a 0 by value to indicate the place of the omitted argument. If the comma appears inside the brackets ([,optional-argument]), you can omit the argument if it is the last argument in the list.

13.7.5 Symbol Definitions

Many system routines depend on values that are defined in separate symbol definition files. VMS RTL routines require you to include symbol definitions when you are calling a Screen Management facility routine or a routine that is a jacket to a system service. A *jacket* routine provides an interface to the corresponding system service. For example, the routine LIB\$SYS_ASCTIM is a jacket routine for the \$ASCTIM system service.

If you are calling a system service, you must include the file SSDEF to check the status. Many system services require other symbol definitions as well. To determine whether you need to include other symbol definitions for the system service you want to use, see the documentation for that particular system service. If the documentation states that values are defined in a macro, you must include those symbol definitions in your program.

For example, the description for the flags parameter in the SYS\$MGBLSC (Map Global Section) system service states that “Symbolic names for the flag bits are defined by the \$SECDEF macro.” Therefore, when you call SYS\$MGBLSC you must include the definitions provided in the \$SECDEF macro.

In VAX C, a definition file is included as follows:

```
#include stdlib
```

For a list of all VAX C definition modules, see Appendix A.

13.7.6 Condition Values

Many system routines return a condition value that indicates success or failure; this value can be either returned or signaled. If a condition value is returned, then you must check the returned value to determine whether the call to the system routine was successful. Otherwise, the condition value is signaled to your program instead of being written to a storage location.

Condition values indicating success appear first in the list of condition values for a particular routine, and success codes have odd values. A success code that is common to many system routines is the condition value `SS$_NORMAL`, which indicates that the routine completed normally and successfully. If the condition value is returned, then you can test for `SS$_NORMAL` as follows:

```
if (ret_status != SS$_NORMAL)
    LIB$STOP();
```

Because all success codes have odd values, you can check a return status for any success code. For example, you can cause execution to continue only if a success code is returned by including the following statements in your program:

```
if ((ret_status & 1) != 0)
    LIB$STOP (ret_status);
```

In general, you can check a return status for a particular success or failure code or you can test the condition value returned against all success codes or all failure codes.

13.7.7 Checking System Service Return Values

It is customary in VMS programming to compare the return status of a system service with a global symbol, not with the literal value associated with a particular return status. Consequently, a high-level language program should define the possible return status values for a service as symbolic constants. In VAX C, you can do this by including the *ssdef* definition module; Example 13–19 shows how this is done.

The system service return status values (`SS$_WASSET` and `SS$_WASCLR`) in Example 13–19 are defined by the *ssdef* definition module.

Example 13–19: Checking System Service Return Values

```
/* This program shows how to compare the status of a system *
 * service with a global symbol. */
/* Define system service *
 * status values */
#include ssdef
#include stdio
/* Declaration of the *
 * service (not required) */
int SYS$SETEF();
main()
{
/* To hold the status of *
 * SYS$SETEF */
int efstatus;
/* Argument values for *
 * SYS$SETEF */
enum cluster0
{
completion, breakdown, beginning
} event;
.
.
event = completion;
/* Set the event flag */
efstatus = SYS$SETEF(event);
/* Test the return status */
if (efstatus == SS$_WASSET)
fprintf (stderr, "Flag was already set\n");
else
if (efstatus == SS$_WASCLR)
fprintf(stderr, "Flag was previously clear\n");
else
fprintf(stderr,
"Could not set completion event flag.\n \
Possible programming error.\n");
exit(efstatus);
}
```

Error handling in Example 13–19 is typical of programs running on VMS systems. Using the following statements, the example program attempts to provide a program-specific error message and then passes the offending error status to the caller:


```

else
    fprintf(stderr,
        "Could not set completion event flag.\n \
Possible programming error.\n");
exit(efstatus);

```

If you execute the program with DCL, it interprets any status value the program returns. DCL prints a standard error message on the terminal to provide you with more information about the failure. For example, if the program encounters the SS\$_ILLEFC return status, DCL displays the following messages:

```

Could not set completion event flag.
Possible programming error.
%SYSTEM-F-ILLEFC, illegal event flag cluster.

```

13.8 Variable-Length Argument Lists in System Services

Most system services and other external procedures require a specific number of arguments, but some accept a variable number of optional arguments. Because VAX C function declarations do not show the number of parameters expected by external functions unless a function prototype is used, the way you call an external function from a VAX C program depends on the semantics of the called function. You must supply the number of arguments that the external function expects. The rules are as follows:

- When optional arguments occur between required arguments, they cannot be omitted. If omitting such an argument is necessary—for example, to select a default action—the argument must be written as a zero.
- When optional arguments occur at the end of an argument list, the format of the function reference depends on the action of the called function as follows:
 - If the called function checks the number of arguments passed, you can omit optional trailing arguments from the function reference. System services generally do not check the length of the argument list.
 - If the called function does not check the number of arguments passed, all arguments must be present in the function reference.

For example, the function STR\$CONCAT, in the Common Run-Time Library, concatenates from 2 to 254 strings into a single string. Its call format is as follows:

```
ret = STR$CONCAT(dst, src1, src2[, src3, ... src254]);
```

For more information about the STR\$CONCAT function, see the *VMS Run-Time Library Routines Volume*.

The identifier `dst` is the destination for the concatenated string, and `src1`, `src2`, . . . `src254` are the source strings. All arguments are passed by descriptor. All but the first two source strings are optional. The function checks to see how many arguments are present in the call; if fewer than three (the destination and two sources) are present, the function returns an error status value. Example 13–20 shows a call to the STR\$CONCAT function from VAX C.

Example 13–20: Using Variable-Length Argument Lists

```
/* This example shows a call to STR$CONCAT. */
#include stdio
#include descrip
#include ssdef

int STR$CONCAT();

main()
{
    int ret;                /* Return status of          *
                           * STR$CONCAT              */

                           /* Destination array of    *
                           * concatenated strings          */
    char dest[21];

                           /* Create compile-time     *
                           * descriptors:                  */
    static $DESCRIPTOR(dst, dest);
    static $DESCRIPTOR(src1, "abcdefghij");
    static $DESCRIPTOR(src2, "klmnopqrst");

                           /* Concatenate strings          */
    ret = STR$CONCAT(&dst, &src, &src2);

                           /* Test return status value */
    if (ret != SS$ _NORMAL)
        fprintf(stderr, "Failed to concatenate strings.\n"),
            exit(ret);

                           /* Process string              */
    else
        dest[20] = '\0',
        printf("Resultant string: %s\n", dest);
}
```

13.9 Return Status Values

The status values from VMS system service procedures are returned in general register R0. This return status value indicates the success or failure of the operation performed by the called procedure. In VAX C, passing a return status value in R0 is equivalent to a function returning **int**.

To obtain a return status value from any system procedure, declare the procedure as a function, as shown in the following example:

```
int  SYS$SETEF();
```

After declaring a procedure in this way, you can invoke the procedure as a function and obtain a return status value. In VAX C, such a declaration is needed only as program documentation; SYS\$SETEF can be called without explicit declaration and will be interpreted by default as a function returning **int**.

This section describes the following topics:

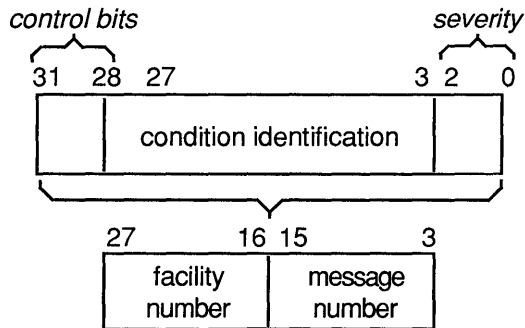
- The format of a return status value, that is, the meaning of particular bits within the value
- The way to manipulate return status values
- The recommended techniques for testing a return status value for success or failure or for a specific condition

13.9.1 Format of Return Status Values

All VMS system procedures and programs use a longword value to communicate return status information. When a VAX C main function executing under the control of the DCL interpreter executes a **return** statement to return control to the command level, the command interpreter uses the return status value to conditionally display a message on the current output device.

To provide a unique means of identifying every return condition in the system, bit fields within the value are defined as shown in the bit fields of Figure 13-5.

Figure 13-5: Bit Fields Within a Return Status Value



ZK-0283-GE

The following list describes the division of the bit field:

control bits (31-28)

Define special action(s) to be taken. At present, only bit 28 is used. When set, it inhibits the printing of the message associated with the return status value at image exit. Bits 29 through 31 are reserved for future use by DIGITAL and must be 0.

facility number (27-16)

Is a unique value assigned to the system component, or facility, that is returning the status value. Within this field, bit 27 has a special significance. If bit 27 is clear, the facility is a DIGITAL facility: the remaining value in the facility number field is a number assigned by the operating system. If bit 27 is set, the number indicates a customer-defined facility.

message number (15-3)

Is an identification number that specifically describes the return status or condition. Within this field, bit 15 has a special significance. If bit 15 is set, the message number is unique to the facility issuing the message. If bit 15 is clear, the message is issued by more than one system facility.

severity (2-0)

Is a numeric value indicating the severity of the return status. The possible values in these three bits, and their meanings, are shown in Table 13-11.

Table 13–11: Possible Severity Values

Value	Meaning
0	Warning
1	Success
2	Error
3	Informational
4	Severe error, FATAL
5-7	Reserved

Odd values indicate success (an informational condition is considered a successful status) and even values indicate failures (a warning is considered an unsuccessful status).

The following names are associated with these fields:

control bits bit 28 (inhibit message)	CONTROLINHIB_MSG
facility number bit 27 (customer facility)	FAC_NOCUST_DEF
message number bit 15 (facility specific)	MSG_NOFAC_SP
severity bit 0 (success)	SEVERITYSUCCESS

When testing return values in a VAX C program, either you can test only for successful completion of a procedure or you can test for specific return status values.

13.9.2 Manipulating Return Status Values

You can construct a structure or union that describes a return status value, but in practice this method of manipulating return status values is not recommended. A status value is usually constructed or checked using bitwise operators. VAX C provides the **#include** module *stsdef*, which contains preprocessor definitions to make this job easier. All of the preprocessor symbols are named according to the VMS naming convention, as follows:

STS\$type_name

STS

Identifies standard return status values.

type

Is one of the following characters denoting the type of the constant:

K	Represents a constant value
M	Represents a bit mask
S	Represents the bit size of a field
V	Defines the bit offset to the field

name

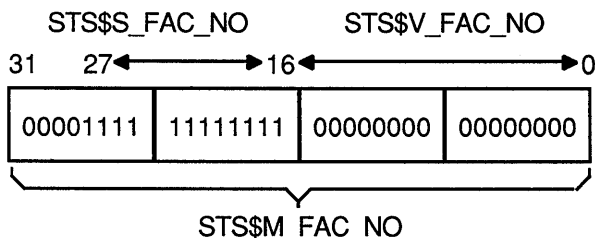
Is an abbreviation for the field name.

For example, the following constants are defined in *stsdef* for the facility number field, FAC_NO, which spans bits 16 through 27:

```
#define STS$S_FAC_NO 12          /* Size of field in bits */
                                /* Bit offset to the      */
                                /* beginning of the field */
#define STS$V_FAC_NO 16
                                /* Bit mask of the field */
#define STS$M_FAC_NO 0xFFF0000
```

Figure 13–6 shows how the status value is represented internally.

Figure 13–6: Internal Representation of a Status Value



ZK-0528-GE

Use the following expression to extract the facility number from a particular status value contained in the variable named status:

```
(status & STS$M_FAC_NO) >> STS$V_FAC_NO
```

In the previous example, the parentheses are required for the expression to be evaluated properly; the relative precedence of the bitwise AND operator (&) is lower than the precedence of the binary shift operator (>>).

13.9.3 Testing for Success or Failure

To test a return status value for success or failure, you need only test the success bit. A value of true in this bit indicates that the return value is a successful value.

Example 13–21 shows a program that checks the success bit.

Example 13–21: Testing for Success

```
/* This program shows how to test the success bit.          */
#include stdio
#include descrip
#include stsdef
main()
{
    int status;
    $DESCRIPTOR(name, "student");
    status = SYS$SETPRN(&name);
    if (status & STS$M_SUCCESS)
        /* Success code */
        fprintf(stderr, "Successful completion");
    else
        /* Failure code */
        fprintf(stderr, "Failed to set process name.\n"),
        exit(status);
}
```

The failure code in Example 13–21 causes the printing of a program-specific message indicating the condition that caused the program to terminate. The error status is passed to the DCL by the **exit** function, which then interprets the status value.

13.9.4 Testing for Specific Return Status Values

Each numeric return status value defined by the system has a symbolic name associated with it. The names of these values are defined as system global symbols, and you can access their values by referring to their symbolic names.

The global symbol names for VMS return status values have the following format:

facility\$_code

facility

Is an abbreviation or acronym for the system facility that defined the global symbol.

code

Is a mnemonic for the specific status value.

Table 13–12 shows some examples of facility codes used in global symbol names.

Table 13–12: Facility Codes

Facility	Description
SS	System services; these status codes are listed in the <i>VMS System Services Volume</i> .
RMS	File system procedures; these status codes are listed in the <i>VMS Record Management Services Manual</i> .
SOR	SORT procedures; these status codes are listed in the <i>VMS Sort / Merge Utility Manual</i> .

The definitions of the global symbol names for the facilities listed are located in the default VAX C object module libraries, so they are automatically located when you link a VAX C program that references them.

When you write a VAX C program that calls system procedures and you want to test for specific return status values using the symbol names, you must perform the following tasks:

1. Determine, from the documentation of the procedure, the status values that can be returned, and choose the values for which you want to provide specific tests.

2. Declare the symbolic name for each value of interest. The *ssdef* and *rmsdef* **#include** modules define the system service and RMS return status values, respectively. If you are checking return status values from other facilities, such as the SORT utility, you must explicitly declare the return values as **globalvalue int**. Consider the following example:

```
globalvalue int SOR$_OPENIN;
```

3. Reference the symbols in your program.

Example 13–22 shows a program that checks for specific return status values defined in the *ssdef* module.

Example 13–22: Testing for Specific Return Status Values

```
/* This program checks for specific return status values. */
#include ssdef
#include stdio
#include descrip

$DESCRIPTOR(message, "\07**Lunch_time**\07");

main()
{
    int status = SYS$BRDCST(&message,0);
    if (status != SS$_NORMAL)
    {
        if (status == SS$_NOPRIV)
            fprintf(stderr, "Can't broadcast; requires OPER \
privilege.");
        else
            fprintf(stderr, "Can't broadcast; some fatal \
error.");
        exit(status);
    }
}
```

13.10 Examples of Calling System Routines

This section provides complete examples of calling system routines from VAX C. Example 13–23 shows the three mechanisms for passing arguments to system services and also shows how to test for status return codes. Example 13–24 shows various ways of testing for successful \$QIO completion. Example 13–25 shows how to use time conversion and set timer routines.

In addition to the examples provided here, the *VMS Run-Time Library Routines Volume* and the *VMS System Services Reference Manual* also provide examples for selected routines. See these manuals for help on using a specific system routine.

Example 13-23: Passing Arguments to System Services

```
/* GETMSG.C
   This program is an example showing the three mechanisms
   for passing arguments to system services. It also
   shows how to test for specific status return
   codes from a system service call. */

#include stdio
#include descrip
#include sdef

main()
{
  int message_id;
  short message_len;
  char text[133];
  $DESCRIPTOR(message_text, text);
  register status;

  while (printf("\nEnter a message number <CTRL/Z to quit>: "),
         scanf("%d", &message_id) != EOF)
  {
    /* Retrieve message associated with the number. */
    status = SYS$GETMSG(message_id, &message_len,
                       &message_text, 15, 0);

    /* Check for status conditions. */
    if (status == SS$NORMAL)
      printf("\n%.*s\n", message_len, text);
    else if (status == SS$BUFFEROVF)
      printf("\nBUFFER OVERFLOW -- Text is: %.*s\n",
            message_len, text);
    else if (status == SS$MSGNOTFND)
      printf("\nMESSAGE NOT FOUND.\n");
    else
    {
      printf("\nUnexpected error in $GETMSG call.\n");
      LIB$STOP(status);
    }
  }
}
```

Example 13-24: Determining \$QIO Completion

```
/* ASYNCH.C
   This program shows various ways to determine
   $QIO completion. It also shows the use of an
   IOSB to obtain information about the I/O operation. */

#include iodef
#include ssdef
#include descrip

typedef struct
{
    short cond_value;
    short count;
    int info;
} io_statblk;

main()
{
    char text_string[] = "This was written by the $QIO.";
    register status;
    short chan;
    io_statblk status_block;
    int AST_PROC();
    $DESCRIPTOR (terminal, "SYS$COMMAND");

    /* Assign I/O channel. */
    if ((status = SYS$ASSIGN (&terminal, &chan, 0, 0)) & 1) != 1)
        LIB$STOP (status);

    /* Queue the I/O. */
    if (((status = SYS$QIO (1, chan, IO$WRITEVBLK, &status_block,
        AST_PROC, &status_block, text_string,
        strlen(text_string), 0, 32, 0, 0)) & 1) != 1)
        LIB$STOP (status);

    /* Wait for the I/O operation to complete. */
    if (((status = SYS$SYNCH (1, &status_block)) & 1) != 1)
        LIB$STOP (status);
    if ((status_block.cond_value & 1) != 1)
        LIB$STOP(status_block.cond_value);

    printf ("\nThe I/O operation and AST procedure are done.");
}

AST_PROC (write_status)
io_statblk *write_status;

/* This function is called as an AST procedure. It uses
   the AST parameter passed to it by $QIO to determine
   how many characters were written to the terminal. */
```

(continued on next page)

Example 13-24 (Cont.): Determining \$QIO Completion

```
{
printf("\nNumber of characters output is %d", write_status->count);
printf("\nI/O completion status is %d", write_status->cond_value);
}
```

Example 13-25: Using Time Routines

```
/* ALARM.C
   This program shows the use of time conversion
   and set timer routines. */

#include stdio
#include descrip
#include ssdef

main()
{
#define event_flag 2
#define timer_id 3

typedef int quadword[2];

quadword delay_int;
$DESCRIPTOR(offset, "0 :::15.00");
char cur_time[24];
$DESCRIPTOR(cur_time_desc, cur_time);
int i;
unsigned state;
register status;

/* Convert offset from ASCII to binary format. */
if (((status=SYSS$BINTIM(&offset, delay_int)) &1) != 1)
    LIB$STOP(status);

/* Output current time. */
if (((status=LIB$DATE_TIME(&cur_time_desc)) &1) != 1)
    LIB$STOP(status);
cur_time[23] = '\0';
printf("The current time is : %s\n", cur_time);

/* Set the timer to expire in 15 seconds. */
if (((status=SYSS$SETIMR(event_flag, &delay_int,
                        0, timer_id)) &1) != 1)
    LIB$STOP(status);
```

(continued on next page)

Example 13-25 (Cont.): Using Time Routines

```
/* Count to 1000000. */
printf("beginning count . . . .\n");
for (i=0; i<=1000000; i++)
    ;

/* Check if the timer expired. */
switch (status = SYS$READEF(event_flag, &state))
{
    case SS$_WASCLR : /* Cancel timer */
        if (((status=SYS$CANTIM(timer_id, 0)) &1) != 1)
            LIB$STOP(status);
        printf("Count completed before timer expired.\n");
        printf("Timer canceled.\n");
        break;
    case SS$_WASSET : printf("Timer expired before count completed.\n");
        break;
    default          : LIB$STOP(status);
        break;
}
}
```

VAX C Implementation Notes

This chapter discusses VAX C program sections.

14.1 Program Sections

The following sections describe program section attributes and program sections created by VAX C.

14.1.1 Attributes of Program Sections (Psects)

As the VAX C compiler creates an object module, it groups data into contiguous program sections, or *psects*. The grouping depends on the attributes of the data and on whether the psects contain executable code or read/write variables.

The compiler also writes into each object module information about the program sections contained in it. The linker uses this information when it binds object modules into an executable image. As the linker allocates virtual memory for the image, it groups together program sections that have similar attributes.

Table 14–1 lists the attributes that can be applied to program sections.

Table 14–1: Program Section Attributes

Attribute	Meaning
PIC or NOPIC	The program section or the data these attributes refers to does not depend on any specific virtual memory location (PIC), or else the program section depends on one or more virtual memory locations (NOPIC). ¹
CON or OVR	Concatenates the program section with other program sections with the same name (CON) or overlays it on the same memory locations (OVR).
REL or ABS	The data in the program section can be relocated within virtual memory (REL) or is not considered in the allocation of virtual memory (ABS).
GBL or LCL	The program section is part of one cluster, is referenced by the same program section name in different clusters (GBL), or is local to each cluster in which its name appears (LCL).
EXE or NOEXE	The program section contains executable code (EXE) or does not contain executable code (NOEXE).
WRT or NOWRT	The program section contains data that can be modified (WRT) or data that cannot be modified (NOWRT).
RD or NORD	These attributes are reserved for future use.
SHR or NOSHR	The program section can be shared in memory (SHR) or cannot be shared in memory (NOSHR).
USR or LIB	These attributes are reserved for future use.
VEC or NOVEC	The program section contains privileged change mode vectors (VEC) or does not contain those vectors (NOVEC).

¹VAX C programs can be bound into PIC or NOPIC shareable images. NOPIC occurs if declarations such as the following are used: "char *x = &y;". This statement relies on the address of variable y to determine the value of the pointer, x.

14.1.2 Program Sections Created by VAX C

If necessary, VAX C creates the following program sections:

- **\$CODE**—Contains all executable code and constant data (including variables defined with the **readonly** modifier).
- **\$DATA**—Contains all static variables, as well as global variables defined without the **readonly** modifier.

- `$CHAR_STRING_CONSTANTS`—Contains VAX C character-string constants (strings of characters delimited by quotation marks ("")) used in the program, such as the following:

```
char *y = "This is a character-string constant *****"
    /* or . . . */
printf("The answer is . . . %d\n", x);
```

- VAX C also creates additional program sections for external variables, and for global variables when you specify a program section name in the global declaration.

All program sections created by VAX C have the attributes PIC, REL, RD, USR, and NOVEC. The `$CODE` psect is aligned on byte boundaries while all other program sections generated by VAX C are aligned on longword boundaries. The `$CHAR_STRING_CONSTANTS` psect has the same attributes as `$DATA`.

Table 14–2 and Table 14–3 summarize the differences in the psects created by VAX C. Table 14–2 assigns a number to all the possible combinations of storage-class specifiers and modifiers. Table 14–3 presents the psect name and the psect attributes for each combination.

Table 14–2: Combinations of Storage-Class Specifiers and Modifiers

Storage Class Code	Storage-Class Keyword Combination
1	<code>[extern]</code>
2	<code>[extern] const readonly</code>
3	<code>[extern] noshare</code>
4	<code>[extern] const readonly noshare</code>
5	<code>static</code>
6	<code>static const readonly</code>
5	<code>static noshare</code>
5	<code>globaldef</code>
6	<code>globaldef const readonly</code>
5	<code>globaldef noshare</code>

(continued on next page)

Table 14–2 (Cont.): Combinations of Storage-Class Specifiers and Modifiers

Storage Class Code	Storage-Class Keyword Combination
7	globaldef {"name"}
8	globaldef {"name"} const readonly
9	globaldef {"name"} noshare
10	globaldef {"name"} const readonly noshare

The numbers in Table 14–2 correspond to the numbers in Table 14–3. In Table 14–3, the [**extern**] *name* psect is the same name of the identifier in the declaration, but the **globaldef** "name" psect can be any name you specify in the {"name"} portion of the declaration.

Table 14–3: Combination Attributes

Storage Class Code	Program Section Name	Program Attributes
1	<i>name</i>	OVR, GBL, SHR, NOEXE, WRT
2	<i>name</i>	OVR, GBL, SHR, NOEXE, NOWRT
3	<i>name</i>	OVR, GBL, NOSHR, NOEXE, WRT
4	<i>name</i>	OVR, GBL, NOSHR, NOEXE, NOWRT
5	\$DATA	CON, LCL, NOSHR, NOEXE, WRT
6	\$CODE	CON, LCL, SHR, EXE, NOWRT
7	"name"	CON, GBL, SHR, NOEXE, WRT
8	"name"	CON, GBL, SHR, NOEXE, NOWRT
9	"name"	CON, GBL, NOSHR, NOEXE, WRT
10	"name"	CON, GBL, NOSHR, NOEXE, NOWRT

NOTE

VAX C creates psects with different attributes for global and external storage classes. See Section 9.6.1.1 for information on the differences between these two storage classes.

The combined use of the **readonly** and **noshare** modifiers is illegal in the following declarations:

```
readonly noshare static int x;  
readonly noshare globaldef int x;
```

When it encounters a situation as shown in the previous example, the compiler ignores the **noshare** modifier and accepts **readonly**. The order of the storage-class specifier, the storage-class modifier, and the data-type keyword within a declaration is not significant.

The VAX C compiler does static (global) initialization of pointers by using the **.ADDRESS** directive. By using this mechanism, the compiler efficiently generates position-independent code. The linker makes image sections that contain such initialization nonshareable.

VAX C Definition Modules

This appendix lists the library definition modules contained in the text library named SYS\$LIBRARY:VAXCDEF.TLB.

You can examine the contents of these modules in the appropriate definition file. All definition files have the file extension .H and are contained in the directory SYS\$LIBRARY. You can print or type individual files, or you can issue the following command to print all the files with their file names appearing at the top of each page:

```
$ PRINT SYS$LIBRARY:*.H/HEADER
```

Table A-1 describes each of the definition modules.

Table A-1: VAX C Definition Modules

Module	Description
<i>accddef</i>	Accounting file record definitions.
<i>acedef</i>	Access control list entry structure definitions.
<i>acldef</i>	Access control list definitions.
<i>acrdef</i>	Accounting record definitions.
<i>argdef</i>	Argument descriptors definitions.
<i>armdef</i>	Access rights definitions.
<i>assert</i>	Assert macro definition.
<i>atrdef</i>	File attribute definitions.
<i>basdef</i>	Message definitions for BASIC.

(continued on next page)

Table A-1 (Cont.): VAX C Definition Modules

Module	Description
<i>brkdef</i>	Breakthrough system service definitions.
<i>chfdef</i>	Structure definitions for condition handlers.
<i>chkpntdef</i>	Flags for calls to create checkpointable processes.
<i>chpdef</i>	Definitions for the \$CHKPRO (check protection) service.
<i>clidef</i>	Command-language interface definitions.
<i>climsgdef</i>	Command-language interpreter error code definitions.
<i>cliservdef</i>	CLI service request codes.
<i>cliverbdef</i>	CLI generic codes for verbs.
<i>clsdef</i>	Security classification mask block definitions.
<i>cobdef</i>	Message definitions for COBOL.
<i>cqualdef</i>	Qualifier definitions.
<i>crdef</i>	Card reader status bits definitions.
<i>credef</i>	Create options table definitions.
<i>crfdef</i>	CRF\$INSRTREF argument list definitions.
<i>crfmsg</i>	Return status codes for cross-reference program.
<i>ctype</i>	Character type and macro definitions for character classification functions.
<i>curses</i>	Curses Screen Management-related definitions.
<i>dedef</i>	Device class and type code definitions.
<i>descrip</i>	Descriptor structure and constant definitions.
<i>devdef</i>	Device characteristics definitions.
<i>dibdef</i>	Device information block definitions.
<i>dmpdef</i>	Layout of the header block of the system dump file.
<i>dmtdef</i>	\$DISMOU (dismount) system service definitions.
<i>dstdef</i>	Debug Symbol Table definitions.
<i>dtk\$routines</i>	DECTalk routine definitions.
<i>dtkdef</i>	Definitions for RTL DECTalk Management.
<i>dtkmsg</i>	Message definitions for DECTalk.
<i>dvidef</i>	\$GETDVI system service request code definitions.

(continued on next page)

Table A-1 (Cont.): VAX C Definition Modules

Module	Description
<i>envdef</i>	Define/reference environment definitions.
<i>eomdef</i>	End-of-module record (EOM) definition.
<i>eomwdef</i>	End-of-module record with word of psect (EOMW) definition.
<i>epmdef</i>	GSD entry - Entry point definition, normal symbols.
<i>epmndef</i>	GSD entry - Entry point definition, version mask symbols.
<i>epmvdef</i>	GSD entry - Entry point definition, vectored symbols.
<i>epmwdef</i>	GSD entry - Entry point definition with word of psect value.
<i>eradef</i>	Erase type codes definitions.
<i>errno</i>	Error number definitions.
<i>errnodef</i>	VAX C error message constants.
<i>fab</i>	File access block definitions.
<i>faldef</i>	Message definitions for the FAL (DECnet File Access Listener).
<i>fchdef</i>	File characteristics definitions.
<i>fdldef</i>	FDL call interface definitions.
<i>fibdef</i>	File information block definitions.
<i>fiddef</i>	FID (File ID) structure definitions.
<i>file</i>	Symbol definitions for the open function.
<i>float</i>	Macro definitions that provide implementation-specific, floating-point restrictions.
<i>fmldef</i>	Formal arguments structure definitions.
<i>fordef</i>	Message definitions for FORTRAN.
<i>fscndef</i>	SYS\$FILESCAN descriptor codes.
<i>gpsdef</i>	GSD entry - Psect definition.
<i>gsdef</i>	Global symbol definition record (GSD) definitions.
<i>gsydef</i>	GSD entry - Symbol definition.
<i>hlpdef</i>	Definitions for help processing.
<i>iacdef</i>	Image activation control flags definitions.
<i>idcdef</i>	Random entity ident consistency check definitions.
<i>iodef</i>	I/O function code definitions.

(continued on next page)

Table A-1 (Cont.): VAX C Definition Modules

Module	Description
<i>jbcmgdef</i>	Message definitions for Job Controller.
<i>jpidef</i>	\$GETJPI system service request code definitions.
<i>kgbdef</i>	Key Grant Block definitions for rights data base.
<i>ladef</i>	LPA-11 characteristics definitions.
<i>latdef</i>	Message definitions for the LAT facility.
<i>lbrctltbl</i>	Library control table use by Librarian.
<i>lbrdef</i>	Librarian argument definitions.
<i>lckdef</i>	Lock manager definitions.
<i>lepmdef</i>	GSD entry - Module local entry point definition.
<i>lhidef</i>	Library header information array offsets.
<i>lib\$routines</i>	Library (LIB\$) routine definitions.
<i>libclidef</i>	Definitions for LIB\$ CLI callback procedures.
<i>libdcfdef</i>	Definitions for LIB\$DECODE_FAULT.
<i>libdef</i>	Definitions of LIB\$ return codes.
<i>libdtdef</i>	Interface definitions for LIB\$DT (date/time) package.
<i>libvmdef</i>	Interface definitions for LIB\$VM package.
<i>limits</i>	Macro definitions that provide implementation-specific constraints.
<i>lkidef</i>	Lock information data identifier information.
<i>lmfdef</i>	License Management Facility definitions.
<i>lnkdef</i>	Linker Options Record (LNK).
<i>lnmdef</i>	Logical name flag definitions.
<i>lpdef</i>	Line printer characteristics definitions.
<i>lprodef</i>	GSD entry - Module Local Procedure definition.
<i>lsdfdef</i>	Module-local Symbol definition.
<i>lsrfdef</i>	Module-local Symbol reference.
<i>lsydef</i>	LSY - Module-local symbol definition.
<i>maildef</i>	Definitions needed for callable mail.
<i>math</i>	Math function definitions.

(continued on next page)

Table A-1 (Cont.): VAX C Definition Modules

Module	Description
<i>mhddef</i>	Object module header definitions.
<i>mhdef</i>	Module header record (MHD).
<i>mntdef</i>	Flag bits for the \$MOUNT system service.
<i>msgdef</i>	System mailbox message type definitions.
<i>mt2def</i>	Extended magtape characteristics definitions.
<i>mtadef</i>	Magtape accessibility routine code definitions.
<i>mtdef</i>	Magtape status definitions.
<i>mthdef</i>	Message definitions for the math library.
<i>nam</i>	Name block definitions.
<i>ncs\$routines</i>	Definitions for routines for working with national character sets.
<i>ncsdef</i>	Message definitions for the NCS facility.
<i>nfbdef</i>	DECnet file access definitions.
<i>nsarcdef</i>	Security Auditing record definitions.
<i>objrecdef</i>	Object file record definitions.
<i>opcdef</i>	OPCOM request code definitions.
<i>opdef</i>	Instruction opcode definitions.
<i>oprdef</i>	Operator communications message types and values.
<i>ots\$routines</i>	Common object library routine definitions.
<i>otsdef</i>	Message definitions for common object library.
<i>pccdef</i>	Printer/terminal carriage-control specifiers.
<i>perror</i>	PERROR function-related definitions.
<i>plvdef</i>	Privileged library vector definition.
<i>ppl\$def</i>	Definitions for the RTL Parallel Processing Facility.
<i>ppl\$routines</i>	Routine definitions for the Parallel Processing Facility.
<i>ppldef</i>	Message definitions for the Parallel Processing Facility.
<i>pqldef</i>	Process quota code definitions.
<i>prcdef</i>	Create process (SYS\$CREPRC) system service status flags.
<i>prdef</i>	Processor register definitions.
<i>processes</i>	Prototype definitions for subprocess functions.

(continued on next page)

Table A-1 (Cont.): VAX C Definition Modules

Module	Description
<i>prodef</i>	GSD entry - Procedure definition, normal symbols.
<i>promdef</i>	GSD entry - Procedure definition, version mask symbols.
<i>provdef</i>	GSD entry - Procedure definition, vectored symbols.
<i>prowdef</i>	GSD entry - Procedure definition with word of psect value.
<i>prtdef</i>	Protection field definitions.
<i>prvdef</i>	Privilege mask bit definitions.
<i>psldef</i>	Processor Status Longword definitions.
<i>psmmmsgdef</i>	Message definitions for print symbiont.
<i>pswdef</i>	Processor Status Word definitions.
<i>quidef</i>	Get Queue Information Service (\$GETQUI) definitions.
<i>rab</i>	Record access block definitions.
<i>rmedef</i>	RMS escape definitions.
<i>rms</i>	All RMS structures and return status value definitions.
<i>rmsdef</i>	RMS return status value definitions.
<i>sbkdef</i>	Statistics block definitions.
<i>scrdef</i>	Screen package request types.
<i>sdfdef</i>	Object symbol definitions.
<i>sdfmdef</i>	Object symbol definition for version mask symbols.
<i>sdfvdef</i>	Object symbol definition for vectored symbols.
<i>sdfwdef</i>	Object symbol definition with word of psect value.
<i>secdef</i>	Image section flag bit and match constant definitions.
<i>setjmp</i>	State buffer definition for the setjmp and longjmp functions.
<i>sfdef</i>	Stack call frame definitions.
<i>sgpsdef</i>	GSD entry - Psect definition in a shareable image.
<i>shrdef</i>	Definition file for shared messages.
<i>signal</i>	Signal value definitions.
<i>sjcdef</i>	Send to Job Controller Service (\$SNDJBC) definitions.
<i>smg\$routines</i>	Curses Screen Management facility routine definitions.

(continued on next page)

Table A-1 (Cont.): VAX C Definition Modules

Module	Description
<i>smgdef</i>	Curses Screen Management interface definitions.
<i>smgmmsg</i>	Message definitions for Curses Screen Management Facility.
<i>smgtrmptr</i>	Terminal Capability Pointers for RTL SMG\$ facility.
<i>smrdef</i>	Symbiont manager request codes definitions.
<i>sort\$routines</i>	Sort/Merge routine definitions.
<i>sortdef</i>	Message definitions for Sort/Merge.
<i>srdef</i>	Object symbol reference.
<i>srmddef</i>	Hardware symbol definitions.
<i>ssdef</i>	System service return status value definitions.
<i>starlet</i>	System routine definitions.
<i>stat</i>	STAT and FSTAT function-related definitions.
<i>stdarg</i>	Variable argument list access definitions.
<i>stddef</i>	Common useful definitions.
<i>stdio</i>	Standard I/O definitions.
<i>stdlib</i>	Definitions of miscellaneous C functions.
<i>str\$routines</i>	Routine definitions for dealing with strings.
<i>strdef</i>	Message definitions for VMS string functions.
<i>string</i>	C string function definitions.
<i>stsdef</i>	System service status code format definitions.
<i>sydef</i>	Definitions for the Get System-Wide Information (SYS\$GETSYI) system service.
<i>time</i>	Definitions for the localtime function.
<i>timeb</i>	Definitions for the ftime function.
<i>tirdef</i>	Object file text, information and relocation record (TIR).
<i>tpadef</i>	TPARSE control block definitions.
<i>trmddef</i>	Define symbols for the item list QIO format.
<i>tt2def</i>	Terminal definitions.
<i>ttdef</i>	Terminal definitions.
<i>types</i>	Type definitions.

(continued on next page)

Table A-1 (Cont.): VAX C Definition Modules

Module	Description
<i>uaidef</i>	Get User Authorization Information Data Identifier definitions.
<i>uicdef</i>	Format of UIC - user identification code.
<i>unixio</i>	UNIX I/O functions.
<i>unixio</i>	UNIX I/O emulation functions.
<i>unixlib</i>	Miscellaneous UNIX emulation functions.
<i>unixlib</i>	UNIX emulation functions.
<i>usgdef</i>	Disk usage accounting file produced by ANALYZE/DISK_STRUCTURE utility.
<i>usridef</i>	User image bit definitions.
<i>varargs</i>	Variable argument list access definitions.
<i>xab</i>	Extended attribute block definitions.
<i>xwdef</i>	System definitions for DECnet DDCMP.

VAX C Compiler Messages

This appendix lists the VAX C compiler diagnostic messages. For each message, the appendix gives the mnemonic, the message text, an explanation of the message, and suggested actions to be taken to avoid the message. For more information about the format of the error messages, see Chapter 1.

You can also obtain the compiler diagnostic messages online. To do so, type the following command:

```
$ HELP CC ERROR mnemonic RETURN
```

To receive a list of all the mnemonics, type the following command:

```
$ HELP CC ERROR RETURN
```

Some messages substitute information from the program in the message text. In this appendix, the portion of the text to be substituted is shown as "*****" or *****. If quotes appear around the asterisks, quotes appear in the substituted message.

You can suppress the warning and informational messages with the `/[NO]WARNINGS` qualifier on the `CC` command line. You may want to do this so that the compiler broadcasts only the most severe messages to the terminal. For more information about the `/[NO]WARNINGS` qualifier, see Chapter 1.

ADDRDEPENDENCE, Potential dependence created by use of variable "****" within the expression which inhibited decomposition at loop control variable "****".

Informational: A variable is used in an address dereferencing expression, which can cause a loop-carried dependency. Loop decomposition is inhibited. This message is issued if you specified the `/PARALLEL` qualifier on the CC command line.

User Action: If you are sure that the target memory accessed conflicts with no other memory accessed during loop execution, use the `#pragma ignore_dependency` to specify loop decomposition.

AGGREGATEDEPEND, Variable "****" has subscript expressions which are not the same, which inhibited loop decomposition at loop control variable "****".

Informational: A variable's subscripts are not the same across two references to the variable, indicating a potential dependency. This message is issued if you specified the `/PARALLEL` qualifier on the CC command line.

User Action: If the loop must be decomposed, then rework your algorithm or use the `ignore_dependency` pragma to disregard the dependency.

ANACHRONISM, The "****" operator is an obsolete form, and may not be portable.

Informational: You used an old-style assignment operator such as `+=` or `=*`.

User Action: For the program to be portable, reverse the order of the operator parts. For example, change `+=` to `+=` and `=*` to `*=`. The old-style operators are currently supported by VAX C, but they may not be supported by other C compilers, and they are not guaranteed to be supported in future releases of VAX C.

ARGINVSTRPTR, The `***` argument of "****" built-in function is not a pointer to structure or union with size: 1, 2, or 4 bytes.

Error: A built-in function that takes a **struct** argument was not passed a **struct** of the appropriate size.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments.

ARGLISTOOLONG, Function reference specifies an argument list whose length exceeds the VAX architecture limit.

Error: The size of your argument list in the function call exceeded 255 longwords.

User Action: Rewrite the function definition and function call with a list whose member(s) take less space; for example, by passing floating-point and structure arguments by reference rather than by value. Recall that floating-point arguments occupy two longwords, unless a function prototype is specified using **float**, and that structures passed by value occupy as many longwords as are necessary to contain the whole structure.

ARGNOFLOAT, The *** argument of "****" builtin function may not be floating point. The argument has been converted to an integer.

Warning: An argument to a built-in function has a floating-point type when it should have an integer type.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. If you wish to pass a **float** argument, use an explicit cast.

ARGNOTINTPTR, The *** argument of "****" builtin function is not a pointer to integer.

Error: An argument to a built-in function does not have the required type of pointer to some type of integer.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. Check the arguments for missing address-of operators (&).

ARGNOTLVALUE, The *** argument of "****" builtin function is not an lvalue.

Error: An argument that is required to be an lvalue is a non-lvalue expression.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. Make sure the appropriate arguments are lvalues.

ARGNOTPTRVAL, The *** argument of "****" builtin function is not a pointer.

Error: An argument that is required to be some type of pointer does not have a pointer type.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. Check the arguments for missing "address of" operators (&).

ARGOVERFLOW, Length of the argument list for macro "****" exceeds buffer capacity; overflowing argument(s) considered to be null.

Warning: The total length of the arguments in a macro reference exceeded the compiler's capacity to store the arguments prior to substitution.

User Action: Shorten or eliminate one or more arguments.

ARGREADONLY, The *** argument of "****" builtin function is read-only.

Error: An argument that is used by the function to modify memory is a pointer to const or read-only memory.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. Make sure that arguments that the function uses to change memory point to writeable memory.

ARGSTOOFEW, Argument list for builtin function "****" contains too few arguments; the builtin function is being ignored.

Error: Not all of the required arguments were specified.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments.

ARGSTOOMANY, Argument list for builtin function "****" contains too many arguments; excess arguments ignored.

Warning: A function was called with extra arguments.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments.

ARRAYDEPENDENCE, Variable "****" has subscript expressions which are not the same, which inhibited loop decomposition at loop control variable "****".

Informational: You have an array that contains a loop-carried dependency due to your use of array subscripts. This message is issued if you specified the /PARALLEL qualifier on the CC command line.

User Action: If the loop must be decomposed, use code transformations as described in Section 3.5 to eliminate the dependency or use the **ignore_dependency** pragma to disregard the dependency.

BADCODE, Invalid code generation sequence.

Fatal: An internal compiler error occurred.

User Action: Gather as much information as you can about the conditions in effect when the error occurred, and submit an SPR (see the *VAX C Installation Guide*).

BADPSECT, The program section (psect) specified by this statement has conflicting "nowrite" attributes with another definition of the same program section.

Warning: You specified two or more references to the same program section, and the attributes of the references do not correspond.

For example, this message appears when two **globaldef** definitions exist for the same name, but only one specifies the storage class **readonly**.

User Action: Make all references to a program section consistent.

BITARRAY, The CDD description for "****" specifies that it is an array of bit-fields; it has been converted to a scalar bit-field.

Informational: The compiler generated a declaration of a bit-field whose size is the same as the total size of the original CDD item. (VAX C does not support arrays of bit fields.)

User Action: If the generated declaration is acceptable, no action is required; otherwise, change the CDD description as appropriate.

BITFIELDSIZE, The CDD description for bit-field "****" specifies a size greater than 32; the excess is declared separately.

Informational: VAX C generated a series of bit-field declarations whose total size is the same as that of the original CDD item. (VAX C does not support individual bit fields larger than 32 bits.)

User Action: If the generated declarations are acceptable, no action is required; otherwise, change the CDD description as appropriate.

BOUNDADJUSTED, The CDD description for "****" specifies non-zero-origin dimension bound(s); adjusted to zero-origin.

Informational: VAX C generated a declaration whose bound(s) have been adjusted to start at zero. The generated array had the same number of elements as the original CDD item. (VAX C does not support dimension bounds that do not start at zero.)

User Action: Make sure that subscript expressions in references to this array are offset by the appropriate amounts.

BUGCHECK, Compiler bug check during ****. Submit an SPR with a problem description.

Fatal: An internal error occurred during the specified phase of compilation.

User Action: Gather as much information as possible about the conditions under which the error occurred, including the phase of compilation, and submit an SPR (see the *VAX C Installation Guide*).

BUILTARGCONV, The *** argument of "****" builtin function has been converted from pointer to arithmetic type.

Warning: An argument that should have an integer or floating-point type had a pointer type.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. If you want to pass a pointer argument to an arithmetic argument, use an explicit cast.

CANTINLINECALL, Can't inline this call to "****" as requested because not enough actual parameters are supplied in the call.

Informational: The number of parameters supplied in a call to the function is fewer than the number of formal parameters declared and used in the function. Function calls that do not supply enough parameters will not be expanded inline.

User Action: Change the call so that all necessary parameters are supplied, or eliminate unneeded formal parameters from the function.

CANTINLINECALL, Can't inline this call to "****" as requested because an offset into a by value parameter exceeds size of actual.

Informational: The actual value of a parameter provided in a call was smaller in size than the corresponding formal parameter of the function. Use of the formal parameter requires the full amount of storage. This indicates that the type of the formal parameter does not match the type of the actual value provided in the call.

User Action: Change either the formal parameter or the actual value provided in the call so that the type of the formal parameter matches the type of the actual value.

CANTINLINEPROC, Can't inline "****" as requested because a variable offset into a by value parameter is used.

Informational: A formal parameter is referenced with a run-time variable subscript. This is usually a parameter of type **struct** containing a field that is an array. Functions that use formal parameters in this way will not be expanded inline.

User Action: Pass a pointer to the **struct** instead of the **struct** itself, or remove the pragma that requests that the function be expanded inline.

CANTINLINEPROC, Can't inline "****" as requested because it declares an exception handler.

Informational: It was requested that a function be expanded inline. However, that function declares an exception handler.

Since the function would not have a call frame, it cannot have an exception handler if it is to be expanded inline.

User Action: Eliminate the exception handler, or remove the pragma that requests that the function be expanded inline.

CANTINLINEPROC, Can't inline "****" as requested because it takes the address of a passed by value parameter.

Informational: The function uses operators such as & to take the address of a formal parameter, or uses the *varargs* package. These practices prevent inline expansion of the function because it may store parameters in registers (which have no address) after inline expansion, and because you may have been relying upon the parameters being adjacent to each other in memory, which will not be true after inline expansion.

User Action: If possible, code the function without using the address of the parameter, or if an address is needed, then change the formal parameter to be a pointer to the value. If the *varargs* package is used, then remove the pragma requesting that the function be expanded inline.

CASECONSTANT, Case label value is not a constant expression.

Error: You specified a value in a **case** label that was not a constant.

User Action: Replace the **case** value with a valid constant expression.

CDDATTRIGNORED, The CDD description for "****" specifies the "****" attribute, which is being ignored.

Informational: The CDD record description specifies an attribute for the indicated item that is not supported by VAX C. The compiler ignores the indicated attribute.

User Action: None.

CDDNAMETOOLONG, The CDD identifier name exceeds 31 characters; "name" truncated to "name".

Warning: You specified an identifier name that exceeded 31 characters.

User Action: Shorten or change the identifier name.

CDDTOODEEP, The attributes for the Common Data Dictionary record description "****" exceed the implementation's limit for record complexity.

Error: The indicated record description was too complex for VAX C to generate declarations that could be used.

User Action: Simplify the record description in the CDD.

CMPLXINIT, "****" is too complex to initialize.

Warning: The depth of the indicated aggregate variable exceeded the limit of 32 levels.

User Action: Simplify or correct the initializer list or declaration, or initialize the variable within an assignment statement.

COLMAJOR, The CDD description for "****" specifies that it is a column-major array; it has been converted to a one-dimensional array.

Informational: VAX C generated a declaration for this item with a single dimension. (VAX C supports only row-major arrays.)

The number of elements in the array is the same as the total number of elements in the original array.

User Action: Make sure that you properly compute references to this array.

COMPILEERR, Previous errors prevent continued compilation. Please correct reported errors and recompile.

Fatal: The compiler detected too many errors to continue.

User Action: Correct the errors reported in the previous compiler messages.

COMPLEXINTEXPR, Loop decomposition inhibited due to complex loop initialization expression.

Informational: A loop initialization expression was a function call or other complex expression. This message is issued if you specified the /PARALLEL qualifier on the CC command line.

User Action: If the loop must be decomposed, rewrite the initialization section in the loop.

COMPLEXSTEPEXPR, Loop decomposition inhibited due to complex loop step expression.

Informational: A loop step expression was a function call or other complex expression. This message is issued if you specified the /PARALLEL qualifier on the CC command line.

User Action: If the loop must be decomposed, rewrite the loop expression.

COMPLEXTERMEXPR, Loop decomposition inhibited due to complex loop termination expression.

Informational: A loop termination expression was a function call or other complex expression. This message is issued if you specified the /PARALLEL qualifier on the CC command line.

User Action: If the loop must be decomposed, rewrite the termination condition in the loop.

CONBUILTARG, Constant expression required for "****" argument of "****" builtin function.

Error: Some built-in functions require that certain arguments be constants or expressions that the compiler can evaluate at compile time to produce a constant. If a nonconstant expression is used for any such argument, this error message is issued.

User Action: Replace the offending argument expression with a constant. If the structure of the program requires that the built-in function be called with different values that can only be calculated at run time, consider using a **switch** statement to call the built-in function with different (constant) arguments on the basis of the run-time expression.

CONFLICTDECL, This declaration of "****" conflicts with a previous declaration of the same name.

Warning: The compiler determined that both declarations refer to the same object, yet the two declarations conflict in data-type or storage-class organization.

In addition, for external variables and global symbols, the compiler may detect conflicting storage-class specifiers or identifiers that are spelled the same but consist of letters that are in different cases (the linker converts all external and global names to uppercase letters). If the compiler issues an error message for this reason,

the program may be correct; issuing a message in this instance is a warning against possible programming errors.

User Action: If the declarations refer to the same object, make sure that they specify the same types and organizations. Otherwise, either rename one of the identifiers, or separate the scopes of the declarations.

CONTROLDEPEND, Loop "****" has control dependence which inhibited loop decomposition.

Informational: You have a loop that contains a **goto**, **exit**, or **return** statement. This message is issued if you specified the /PARALLEL qualifier on the CC command line.

User Action: If the loop must be decomposed, rework your algorithm.

CRXCONDITION, Common Data Dictionary description extraction condition.

Informational: An anomaly occurred during the extraction of a CDD record description. The specific condition is described in an accompanying message. The severity of this message may be increased to warning or error depending on the specific condition.

User Action: If necessary, correct the indicated condition in the CDD record description.

DEFTOOLONG, Text in #define preprocessor directive is too long; directive ignored.

Warning: The length of the macro string in the **#define** directive exceeded the implementation's limit.

User Action: Simplify the directive.

DIVIDEZERO, Constant expression includes divide by zero; the result has been replaced with 0.

Warning: A division by zero was encountered in a constant expression. The expression was replaced by 0.

User Action: Make sure that no divisors in the expression can evaluate to 0.

DUPCASE, Duplicate case label value "*****".

Error: You specified more than one **case** for the indicated value in a **switch** statement. (The cases must be unique.)

User Action: Change the **case** labels and combine the cases or both, as appropriate.

DUPDEFAULT, Duplicate default label.

Error: You specified more than one **default** case in the same **switch** statement.

User Action: Combine the cases or make other changes necessary to eliminate the duplicate(s).

DUPDEFINITION, Duplicate definition of "*****".

Warning: The named definition appeared more than once in the program.

The two definitions are essentially the same. Both definitions specify the same data types and organizations, but there may be differences in the values, initializers, or array bounds. If the name is a function, there may be a difference in the number or types of parameters, or in the contents of the function body.

User Action: The purpose of this message is to call a possible programming error to your attention.

DUPINLINEFUNC, Duplicate [no]line function "*****".

Warning: You duplicated a function name in one or more pragma declarations.

User Action: Change the name of the function declaration.

DUPLABEL, Duplicate label "*****".

Error: You specified duplicates of the indicated label in the same function. (Label identifiers must be unique within a function definition.)

User Action: Rewrite the labels (and **goto** statements that refer to them) to eliminate the duplicates.

DUPLISTITEM, Duplicate list item "*****" ignored.

Warning: You specified the same name more than once in a list of arguments to a **#pragma** directive. For example, in the following **#pragma**, the second appearance of variable a is redundant and is ignored:

```
#pragma noinline (a, b, a)
```

Similarly, the second occurrence of variable y in the following example is redundant, as argument lists for some **#pragma** directives are concatenated:

```
#pragma noinline (x, y)
.
.
.
#pragma noinline (y, z)
```

User Action: Remove the duplicate argument if it is redundant; otherwise, check for misspellings.

DUPMAINFUNC, Duplicate main function.

Warning: You defined two or more main functions in a single compilation unit.

A main function is either a function with the name main or a function with the main_program option. If the compilation unit contains more than one main function, the compiler recognizes only the first as the main function.

User Action: Make sure that there is only one main function defined in the compilation unit.

DUPMEMBER, Duplicate declaration of member "*****".

Warning: You declared two members with the same name in the same structure.

User Action: Rename one of the members or remove one of the member declarations.

DUPPARAMETER, Duplicate parameter "*****" ignored.

Warning: The stated function parameter occurred more than once in the function's formal parameter list. For example:

```
funct (a,b,c,a) { }
```

All occurrences of the parameter after the first are ignored.

User Action: Remove or change the duplicate parameter identifier.

ENUMCLASH, Mismatched enum type in "****" operation.

Warning: The indicated operation combined an **enum** variable or value with a value of a nonmatching type. The compiler issued this message if you used the /STANDARD=PORTABLE qualifier on the CC command line.

User Action: Use a cast operation to cast either the **enum** value or the other value to a matching type.

ENUMOP, "****" is an undefined operation for enum values; enum operand(s) converted to int.

Warning: You used an **enum** variable or constant with an arithmetic or bitwise operator. These operators are undefined for use with **enum** types. The operation is performed; however, the compiler treats the **enum** object as an integer.

User Action: Cast the **enum** object to **int**.

EXTERNNAMETOOLONG, The external identifier name exceeds 31 characters; "name" truncated to "name".

Warning: You specified an external identifier name that exceeded 31 characters.

User Action: Shorten or change the external identifier name.

EXTRACOMMA, Extraneous comma in "****" ignored.

Warning: You have coded an extra comma in the indicated context; the comma was ignored by the compiler.

User Action: Make sure that any required item was not accidentally omitted; otherwise, remove the extra comma.

EXTRAFORMALS, Extraneous formal parameter(s) ignored in declaration of "****"

Warning: You included a function's formal parameters in a function declaration or definition.

For example, the following function declaration is not allowed because it names the function's parameters:

```
int funct(a,b,c);
```

The parameters a, b, and c are ignored.

Similarly, the following example defines a function returning a pointer to a function returning an integer. The names of the parameters of the function returning an integer are not allowed.

```
(*f(p1,p2))(q1,q2)
int p1, p2;
{ . . . }
```

The compiler ignores the parameters q1 and q2.

User Action: Check the syntax of the function declaration and, if appropriate, remove the extra identifier(s).

EXTRAMODULE, Redundant #module preprocessor directive ignored.

Warning: You specified more than one #module directive in a single compilation; the excess directive or directives were ignored.

User Action: Make sure that only one #module directive exists in the source file, and that it is placed before any VAX C source code.

EXTRATEXT, Extraneous text in preprocessor directive ignored.

Informational: Extra text appeared in the directive. For example:

```
#endif ABC
```

The compiler issues this message if you specify the /STANDARD=PORTABLE qualifier on the CC command line.

User Action: Either remove the extra text or enclose it in a comment.

FATALSYNTAX, Fatal syntax error.

Fatal: The compiler could not continue due to syntax errors.

User Action: Correct the error on the indicated line and errors, or both, reported in previous compiler messages.

FILENOTFOUND, Include file could not be opened.

Fatal: The compiler could not find the include file in any of the valid text libraries or directories.

User Action: Check to see if the file exists, and then check that the include method you used for this file searches for the file in the place where you expected it to search.

FLOATCONFLICT, The CDD description for "****" specifies the D_floating data type; the data cannot be represented when compiling with /G_FLOAT.

FLOATCONFLICT, The CDD description for "****" specifies the G_floating data type; the data cannot be represented when compiling with /NOG_FLOAT.

Warning: The data type of the indicated CDD item conflicted with the indicated command-line qualifier.

Only one of the two double-precision, floating-point data formats may be used in a compilation, as specified by the command-line qualifier (the default qualifier is /NOG_FLOAT). VAX C generates a declaration of an 8-byte structure for the item.

User Action: Specify the appropriate command-line qualifier, or change the description of the item in the CDD.

FLOATOVERFLOW, Overflow during evaluation of floating-point constant expression.

Error: Overflow occurred during the evaluation of a constant expression containing floating-point operands.

User Action: Make sure that the expression value is in the range $0.29 * 10^{-38}$ to $1.7 * 10^{38}$.

FUNCNOTDEF, Static function "****" is not defined in this compilation; assumed to be external.

Warning: The indicated static function declaration did not refer to an existing definition. The compiler treated the function as external.

User Action: Remove the storage-class specifier **static** in the function declaration or use the specifier in the appropriate function definition.

GLOBALENUM, Enumerators may not be initialized when declared with "globalref".

Warning: You tried to specify the values of enumeration constants in a declaration of an **enum** variable with the **globalref** storage-class specifier.

You must define these values elsewhere, in a **globaldef** declaration, and you must not initialize them in the **globalref** declaration.

User Action: Remove all initializing values from the **globalref** declaration.

IFEVALERROR, **** while evaluating #if or #elif expression; "true" expression assumed.

Warning: The substitute text is "Stack overflow" or "Divide by zero".

User Action: For stack overflow, reduce the complexity of the expression. For divide by zero, make sure that no divisors are zero.

IFSYNTEX, Syntax error in #if or #elif expression; true expression assumed.

Warning: The **#if** or **#elif** expression on the indicated line cannot be evaluated because of syntax errors; it was assumed to be true.

User Action: Correct the line.

IGNORED, Unexpected **** ignored.

Warning: The compiler encountered an unexpected macro in the source program, and has ignored it. (This may be a syntax error.)

User Action: Make sure that the macro and surrounding text is syntactically correct.

INCBUILTARG, Incorrect type for *** argument of "****" builtin function.

Error: An argument to a built-in function has the wrong type.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments.

INLINCONF, Previous inline or noinline pragma for "*****" conflicts with this pragma.

Warning: You used both an inline pragma and a noinline pragma specifying conflicting inline specifications for one particular function.

User Action: Determine whether you want the function to be expanded inline, and remove the conflicting pragma.

INSBEFORE, Inserted **** before ****.

Warning: The compiler tried to recover from a syntax error by inserting a macro into the source.

User Action: Correct the syntax.

INSMATCH, Inserted **** to match **** on line ****.

INSMATCH, Inserted **** to match **** inserted earlier.

Warning: The compiler tried to recover from a syntax error by inserting a macro to match a previous macro in the source code. The previous macro may or may not have been inserted by the compiler.

User Action: Make sure that you match all parentheses, brackets, and braces.

INTERNALLIMIT, Loop decomposition inhibited due to compiler's internal limitations.

Informational: You have too many loops within a single function for decomposition to proceed. This message is issued if you specified the /PARALLEL qualifier on the CC command line.

User Action: If the loop must be decomposed, create two functions in place of the existing function or create nested loops in place of the single loops.

INTVALERROR, Integer value not used where required.

Error: You used a noninteger value as an initializer for an **enum** constant, or to specify the size of a bit field. You must specify these values as integer constants.

User Action: Specify an integer constant.

INTVALREQ, Noninteger value used incorrectly in a **** ; converted to integer.

Warning: You used a noninteger value in a **switch** statement or a **case** label. The value has been converted to integer.

User Action: Specify **switch** expressions and **case** label values as integer values, or use a cast operator to make the conversion explicit.

INVAGGASSIGN, Invalid aggregate assignment.

Error: You tried to assign an array to another array or to assign structures or unions of different sizes.

User Action: Correct the assignment.

INVALIDIF, "*****" is not a valid constant or operator in a #if or #elif expression; "true" expression assumed.

Warning: You used an invalid construction in an #if or #elif expression, which is assumed to be true.

User Action: Correct the expression.

INVALIDNSPEC, Invalid alignment specification ignored.

Warning: You specified an alignment option that was not in the range allowed. The compiler ignored the specified option.

User Action: Correct the alignment specification.

INVALIDINIT, The initialization of "*****" is not valid.

Warning: The indicated object cannot be initialized as specified. Some objects may not be initialized at all, such as functions, unions, and **extern** or **globalref** objects. In other cases, the initializer may not be appropriate; for example, a static pointer cannot be initialized with the address of an automatic variable. This and any subsequent initializers for the same object have been ignored.

User Action: Eliminate or correct the initializer, or correct the type or storage class of the target object, or initialize the object with an explicit assignment.

INVANAFILE, The compiler has generated an invalid ANA file. Please submit an SPR with the sources which generate this error.

Warning: The compiler has generated some invalid data in the ANALYSIS_DATA file.

User Action: Correct all other errors. If the error persists, please submit an SPR.

INVARRAYBOUND, The declaration of "****" specifies a missing or invalid array bound.

Error: In a declaration of an array, you omitted a required dimension bound value or specified an invalid value for a bound.

For multidimensional arrays, you must specify bounds for dimensions other than the first. You must also specify a bound for the first (or only) dimension if this declaration is a definition. Valid bound values are integer constant expressions greater than zero.

User Action: Make sure that all required bounds are present and valid.

INVARRAYDECL, "****" is an improperly declared array.

Error: You improperly declared an array, such as an array of functions.

User Action: Make sure that the syntax of the declarator correctly describes the object. (The declared object may not be what you want.) You may find the output from the /SHOW=SYMBOLS qualifier to be helpful in diagnosing this error.

INVASSIGNTARG, Invalid target for assignment.

Error: You specified, as the left operand of an assignment operator an expression that was not valid for assignment. For example, you may have tried to assign something to an array, to a function, to a constant, or to a variable declared with the **readonly** storage-class modifier.

User Action: Make sure that the target is appropriate for assignments.

INVBREAK, Invalid use of the "break" statement.

Error: You used **break** outside the body of a **for**, a **while**, a **do**, or a **switch** statement.

User Action: Remove the **break** statement, or check that any braces in recent loops or **switch** statements are properly balanced.

INVBUILTIN, The "****" builtin function call is being ignored; it has invalid argument(s).

Error: A call to a built-in function contains errors. This message usually follows other error messages describing errors in the argument expressions.

User Action: Correct any errors listed before this one. Make sure that the function is called with the correct number and types of arguments.

INVCMDVAL, "*****" is an invalid command qualifier value.

Fatal: The indicated CC command qualifier value was acceptable to the VMS command language interpreter (CLI), but it is meaningless to VAX C; for example, LIST_OPTS is an invalid value for /SHOW, but it is accepted by the CLI.

User Action: Correct the qualifier value.

INVCONDEXPR, The second and third operands of a conditional expression cannot be converted to a common type.

Error: You specified an invalid combination of operands in a conditional expression.

This can occur if the operands are pointers to objects of a different size of type, or if the operands are different structures.

User Action: Make sure that both operands are of compatible sizes and data types.

INVCONST, "*****" is an invalid numeric constant.

Warning: The indicated constant contained illegal characters or was otherwise invalid.

User Action: Correct the constant.

INVCONTINUE, Invalid use of the "continue" statement.

Error: You used the **continue** statement outside the body of a **for**, **while**, or **do** statement.

User Action: Remove the **continue** statement, or check that any braces in recent loops are properly balanced.

INVCONVERT, The source or target of a conversion is noncomputational.

Error: One of the operands in an expression could not be converted as specified. For example, you tried to cast some object to a structure.

User Action: Correct the expression or cast.

INVDATATYPE, "****" has an invalid data type for use in this #pragma preprocessor directive; directive ignored.

Warning: The indicated identifier was not declared with the data type required by the directive in which it appears. The entire directive was ignored by the compiler. The following example requires that p and q be variables:

```
#pragma ignore_dependency(p,q)
```

The next example requires that f and pf be either functions or pointers to functions:

```
#pragma safe_call (f,pf)
```

User Action: Make sure that only appropriately declared identifiers appear in these directives.

INVDEFNAME, Missing or invalid name in **** preprocessor directive; directive ignored.

Warning: The indicated directive was missing a required name. For example:

```
#define
```

The entire directive was ignored.

User Action: Correct or remove the directive.

INVDICTPATH, Missing or invalid path name in #dictionary preprocessor directive; directive ignored.

Warning: The indicated directive was missing a required name. For example:

```
#dictionary
```

The compiler ignores the entire directive.

User Action: Correct or remove the directive.

INVFIELD SIZE, The declaration of "****" specifies an invalid field size; size of 32 bits assumed.

Warning: The indicated field declaration was invalid because it specified too large a size.

User Action: Correct the declaration to specify either a single, smaller field or several contiguous fields.

INVFIELDTYPE, The declaration of "****" specifies an invalid data type; type "unsigned" assumed.

Warning: You declared a field with an invalid data type. Fields must be declared (and manipulated) as integers or enumerated types.

User Action: Correct the declaration to specify a valid data type.

INVFILESPEC, Missing or invalid file specification in #include preprocessor directive; directive ignored.

Warning: The #include directive either was missing a file or module name or specified one that is syntactically invalid. The directive was ignored.

User Action: Correct the directive.

INVFUNCDECL, "****" is an improperly declared function.

Error: You improperly declared a function. For example, you may have omitted the parameter list or a semicolon between the function and a previous declaration.

User Action: Correct the syntax of the declaration.

INVFUNCOPTION, Invalid function definition option "****" ignored.

Warning: The indicated function definition was not supported. (The only valid option is the main_program option.)

User Action: Check the spelling of the option, or the syntax of the function definition.

INVHEXCHAR, Invalid hexadecimal character value; high-order bits truncated.

Warning: An escape character specified in hexadecimal exceeded the limit of a 1-byte character.

User Action: Correct the hexadecimal constant to represent a valid escape character.

INVHEXCON, Hexadecimal constant contains an invalid character.

Error: You specified an invalid hexadecimal constant, such as 0xG.

User Action: Correct the constant.

INVIFNAME, Missing or invalid name in #ifdef or #ifndef preprocessor directive; "true" assumed.

Warning: You specified no name, or a syntactically invalid one, in the directive; the result of the test is assumed to be true.

User Action: Correct the directive.

INVINAGGASN, Invalid "****" built-in function call; structure or union arguments are not of same size.

Error: A built-in function that requires two or more arguments be of the same size was called with arguments of different sizes.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments.

INVLINFILE, Invalid file specification in #line preprocessor directive; directive ignored.

Warning: The file specification was syntactically invalid, and the directive was ignored.

User Action: Correct the directive.

INVLINELINE, Missing or invalid line number in #line preprocessor directive; directive ignored.

Warning: The line number was missing or was syntactically invalid, and the directive was ignored.

User Action: Correct the directive.

INVMMAINRETVL, Return value of main function is not an integer type.

Warning: You have declared a main function with a return value that is not an integer type.

User Action: Check for an omitted semicolon at the end of any declaration immediately preceding the declaration of the main function, or change the return value specification to one of the integer types.

INVMODIDENT, Invalid ident specification in #module preprocessor directive; directive ignored.

Warning: The ident specification in the directive either was not a valid identifier or was not a valid character-string constant.

User Action: Correct the directive.

INVMODIFIER, "****" is an invalid data type modifier in this declaration.

Warning: You specified a data-type modifier other than **const** or **volatile** as in the following example:

```
char * int ptr;
```

The data-type modifier **int** will be ignored.

User Action: Remove or change the data-type modifier.

INVMODTITLE, Missing or invalid title specification in #module preprocessor directive; directive ignored.

Warning: The required title in the directive either was missing or was not a valid identifier.

User Action: Correct the directive.

INVOCTALCHAR, Invalid octal character value; high-order bits truncated.

Warning: The octal value in an escape sequence was too large, as in '\477'. Its high-order bits were truncated.

User Action: Correct the value.

INVOPERAND, Invalid **** operand of a "****" operator.

Error: You specified an invalid operand for the indicated operator.

This message is issued for arithmetic and bitwise operators if the operand is noncomputational (such as a structure). For other operators (such as the increment operator), the compiler issues the message if the operand is not an lvalue. For binary operators, the substituted text indicates which operand, left or right, is invalid.

User Action: Make sure that the operand is the proper type for the operator, and that it is an lvalue.

INVPPKEYWORD, Missing or invalid keyword in preprocessor directive; directive ignored.

Warning: You wrote a directive with no keyword. For example:

```
# ABC
```

The directive was ignored.

User Action: Correct or remove the directive.

INVPROTODEF, The parameter list for a function prototype definition must associate an identifier with each type.

Error: The function definition uses the prototype format but does not contain an identifier for each type in the parameter list.

User Action: Place an identifier name in the appropriate type declaration.

INVPTRMATH, Invalid pointer arithmetic.

Error: You tried to perform an invalid arithmetic operation on a pointer or pointers. The only valid arithmetic operations allowed with pointers are addition and subtraction.

For addition, the only forms allowed are as follows:

```
pointer + integer  
pointer += integer
```

For subtraction, the only forms allowed are as follows:

```
pointer - integer  
pointer -= integer  
pointer - pointer
```

In the last form, both pointers must point to objects of the same size.

User Action: Make sure that the expression conforms to one of the previous forms listed. If necessary, cast one or both operands to a compatible type.

INVSTORCLASS, "****" is an invalid storage class in this declaration.

INVSTORCLASS, The "****" storage class is invalid for the declaration of "****".

Warning: You made one of the following programming errors:

- You specified a storage class that is invalid in the context in which the declaration appears; for example, you specified **auto** in a declaration located outside of a function.
- You specified a storage class that is incompatible with another storage-class specifier; for example, you specified both **static** and **extern**.
- You specified a storage class that is incompatible with the data type of the indicated declarator; for example, you specified **globalvalue** for an array.

In all cases, the compiler ignores the storage-class specifier.

User Action: Correct the declaration.

INVSUBUSE, Invalid use of subscripting.

Error: You specified a subscript in reference to a bit field.

User Action: Correct the syntax. If the structure containing the bit field is an array, you must specify the subscript(s) with the qualifier instead of the member name.

INVSUBVALUE, Invalid subscript value.

Error: You specified a subscript value that is not of an integer type.

User Action: Change or cast the value to an integer type.

INVTAGUSE, Invalid use of tag "****".

Error: You used a previously defined tag name in a declaration of a different type. For example:

```
enum    color {red, green, blue};  
struct  color *cp;
```

You may only use a given tag with one of the types **enum**, **struct**, or **union**. Any identifiers declared with the mismatched type will be undefined.

User Action: Either make sure that each use of the tag name specifies the same type, or use different tag names with each type.

INVVARIANT, Invalid declaration of variant aggregate.

INVVARIANT, Invalid declaration of variant aggregate "*****".

Error: You specified a missing or invalid declarator in the declaration of a `*variant_struct*` or `*variant_union*`. For example, you have specified a list of declarators, a declarator of array, a function, or a pointer type.

User Action: Either declare the aggregate as an ordinary `*struct*` or `*union*` or specify a single, simple identifier as the declarator.

INVVOIDUSE, "void" is only valid in a parameter list when it appears alone. Its use is ignored.

Warning: `void` has been used in a function prototype parameter list but is not the only item in the list.

User Action: Either eliminate `void` or eliminate the extra parameter types in the parameter list.

LIBERROR, Error while reading library "*****".

Fatal: The compiler could not read the indicated library. Either it was not a text library, or its format had been corrupted.

User Action: Verify the spelling of the library's name, and verify that it is a valid VMS text library.

LIBLOOKUP, "*****" was not found in any of the specified libraries.

Fatal: The compiler failed to locate the indicated `#include` module in any of the specified or default libraries.

User Action: Check the CC command line to verify that the library containing the module was specified and that the module name, if specified, was spelled correctly. If the library was a default library, verify (with `SHOW TRANSLATION C$LIBRARY`) that its name is the equivalent for `C$LIBRARY`.

LISTTOOLONG, List in `#pragma` preprocessor directive is too long; directive ignored.

Warning: You have specified more than 128 items in the list. The entire directive was ignored by the compiler.

User Action: Split the list into separate directives.

LIVEOUTSIDELOOP, Variable "****" is in use outside the loop, which inhibited loop decomposition at control variable "****".

Informational: You have a scalar variable that has recurrence in a loop or has a lifetime outside of the loop. This message is issued if you specified the /PARALLEL qualifier on the CC command line.

User Action: If the loop must be decomposed, use one of the code-replication techniques described in Section 3.5.2 or use the **sequential_loop** decomposition pragma to specify no decomposition for the loop.

MACDEFINREF, A macro cannot be **** during the scan of a reference to the macro; directive ignored.

Warning: You tried to redefine or undefine a macro within a reference to it. The compiler ignores the preprocessor directive.

User Action: Move the directive to a position outside the macro reference.

MACNONTERMCHAR, Nonterminated character constant in macro argument; apostrophe added at end of line.

Warning: You omitted the closing apostrophe in a character constant appearing in an argument in a macro reference.

User Action: Correct the constant.

MACREQARGS, Macro reference requires an argument list; "*****" not substituted.

Error: You wrote a macro reference without an argument list. The reference was deleted from the source file.

User Action: Correct the reference, specifying the same number of arguments as in the definition of the macro.

MACSYNTAX, Syntax error in macro definition; directive ignored.

Warning: The syntax of the parameter list in a macro definition was invalid. (You must enclose the parameter list in parentheses and delimit individual parameters with commas.)

User Action: Correct the syntax.

MACUNEXPEOF, Unexpected end-of-file encountered in a macro reference; "****" not substituted.

Error: The end-of-file was encountered during a macro reference; the reference was deleted.

User Action: See if you misplaced the closing parenthesis in the macro argument list.

MAXMACNEST, Maximum text replacement nesting level exceeded; "****" not substituted.

Error: You specified a macro reference that is recursive or otherwise causes repeated substitutions to a depth greater than the implementation maximum of 64.

User Action: Correct the recursion or simplify the definitions.

MERGED, Merged **** and **** to form ****.

Warning: The compiler merged two separate source macros into a single macro.

For example, two plus signs separated by a space may be merged to form the increment operator (++).

User Action: If the compiler's action is correct, remove the space between the macros. Otherwise, check for a missing macro between those merged.

MISARGNUMBER, The number of arguments passed to the function does not match the number declared in a previous function prototype.

Warning: The function call contains too few or extra arguments.

User Action: Correct the number of arguments passed to the function. If the prototype is incorrect, correct the prototype.

MISPARAMNUMBER, The number of parameters declared does not match the number declared in a previous function prototype.

Warning: A function prototype for this function, which appeared earlier in the source file, contains a different number of parameters than this declaration.

User Action: Determine which declarator is correct and modify the other declarator to match it.

MISPARAMTYPE, The type of parameter "****" does not match the type declared in a previous function prototype.

Warning: The type of a parameter in a function definition does not match the type specified for that parameter in the previous prototype.

User Action: Determine which type is correct for that parameter and correct either the function definition or the prototype.

MISPARENS, Mismatched parentheses in #if or #elif expression; "true" expression assumed.

Warning: The expression in a #if or #elif preprocessor directive contained unbalanced parentheses.

User Action: Make sure that you balanced the parentheses in the expression.

MISPRAGMASTAND, Mismatched #pragma standard preprocessor directive(s)

Informational: The compiler detected more occurrences of the **nonstandard** pragma than it did the **standard** pragma.

User Action: Check that each **nonstandard** pragma has a matching **standard** pragma, both in the main source file and in any included files.

MISSENDIF, Missing #endif preprocessor directive(s).

Error: The compiler did not encounter an #endif line for the most recent #if, #ifdef, or #ifndef.

User Action: Be sure that all directives are properly structured, and, if appropriate, add the missing #endif preprocessor directive(s).

MISSEXP, Missing or invalid exponent in float constant; zero exponent ('e0') assumed.

Warning: You wrote a floating-point constant with the letter 'e' or 'E' but with no exponent or an invalid exponent. The exponent was assumed to be zero.

User Action: Correct the constant.

MISSPELLED, Replaced **** with ****.

Warning: You misspelled a reserved word.

User Action: Correct the spelling.

MISWIDETYPE, The prototype for this function does not specify the default widened type for the parameter.

Error: A prototype was declared with a parameter having a type that is, by default, widened with old-style function definitions. For example, a **float** is, by default, sized to a **double** for old-style function definitions. If a prototype is in scope with a size of **float**, then the argument will not have the size that the function expects.

User Action: Correct the declaration in the prototype to specify the larger, widened type. If the type is a **float**, then specify **double**.

MODULENAMELONG, Identifier name in #module exceeds 31 characters; "name" truncated to "name".

Warning: You specified an identifier name that exceeded 31 characters.

User Action: Shorten or change the identifier name.

MODZERO, Constant expression includes mod 0; the result has been replaced with 0.

Warning: The constant expression had an invalid mod expression, such as `5 % 0`. The result was 0.

User Action: Correct the expression (but note that its operands must not be floating point).

NAMETOOLONG, Identifier name exceeds 255 characters; truncated to "****".

Warning: VAX C identifiers are limited to a length of 255 recognized characters.

User Action: Shorten the indicated identifier.

NESTEDCOMMENT, Nested comment encountered.

Informational: The compiler detected an opening comment delimiter (/*) within another comment. (VAX C does not support the nesting of comments; the first ending comment delimiter (*/) encountered ends the comment.)

User Action: Check that you have not misplaced a comment delimiter and accidentally "commented out" necessary code.

NOBJECT, No object file produced.

Informational: The compiler did not produce an object file, due to conditions reported in previous messages.

User Action: Make the corrections suggested by the other message(s).

NOFLOATOP, The **** operand of a "*****" operator has been converted from floating-point to integer.

Warning: The compiler converted the operand to an integer.

The left or right operand of the indicated binary operator, or the operand of the indicated unary operator, cannot be of type **float** or **double**.

User Action: Change or cast the operand to an integral type.

NOLIFETIME, Variable "****" has no lifetime information, which inhibited loop decomposition.

Warning: You have a loop that contains too many scalar variables. This message is issued if you specified the /PARALLEL qualifier on the CC command line.

User Action: If the loop must be decomposed, reorder the declarations of your scalar variables so that the variables most important for decomposition are listed first.

NOLISTING, No listing file produced.

Informational: The compiler did not create a listing file (usually due to previously reported errors).

User Action: None.

NOMIXNMATCH, The parameter list of a function can either contain all identifiers or all types, but not both.

Error: The parameter list of a function contains some type specifiers and some identifiers that do not have type specifiers.

User Action: Either eliminate the type specifiers or add type specifiers to the identifiers that are missing them to create a valid function prototype.

NONAUTOMATIC, Variable "****" is not declared automatic, which inhibited loop decomposition at loop control variable "****".

Informational: You have a loop control variable that does not have the **[auto]** storage class. Only **[auto]** variables can be placed in registers and placing scalars in registers is required for decomposition to occur. This message is issued if you specified the **/PARALLEL** qualifier on the CC command line.

User Action: If the loop must be decomposed, declare the loop control variable to be **[auto]**.

NONEXTERN, Variable "****" is not declared external, which inhibited loop decomposition at loop control variable "****".

Informational: You have a scalar variable in a loop that has the **globaldef** or **static** storage class. This message is issued if you specified the **/PARALLEL** qualifier on the CC command line.

User Action: If the loop must be decomposed, change the declaration of the scalar variable to **[auto]** or **[extern]**.

NONOCTALDIGIT, Octal escape sequence in a character or string constant terminated by a nonoctal digit.

Warning: There was an 8 or 9 in the second or third position of an octal escape sequence. In this case, the digits preceding the nonoctal digit were evaluated, and the 8 or 9 was considered a separate character. The compiler issued this message if you used the **/STANDARD=PORTABLE** qualifier on the CC command line.

User Action: Make sure that the escape sequence contains only octal digits. If the 8 or 9 is separate from the escape sequence, but must immediately follow it, then pad the escape sequence to three digits using leading zeros.

NONOCTALESC, Escape sequence in a character or string constant starts with a nonoctal digit.

Warning: The first of three digits of an escape sequence was an 8 or 9. In this case, the backslash is ignored, and the 8 or 9 was treated as a character. The compiler issued this message if you used the /STANDARD=PORTABLE qualifier on the CC command line.

User Action: Make sure that the compiler correctly resolved the ambiguity.

NONPORTADDR, Taking the address of a constant may not be portable.

Informational: You used an ampersand operator with a constant in the argument list of a function call. (VAX C permits this special case, but other compilers may not.)

User Action: If you do not require portability, no action is necessary. Otherwise, correct the line.

NONPORTARG, Passing a structure by value may not be portable.

Informational: You passed a structure by value in a function call or declared a function parameter as a structure. This message is issued if you used the /STANDARD=PORTABLE option on the CC command line.

User Action: If the program must be portable, pass the structure by reference.

NONPORTCLASS, Storage class "****" is not portable.

Informational: This message was issued against the use of the **globalref**, **globaldef**, **globalvalue**, **readonly**, or **noshare** storage-class specifiers. This message is issued if you specified the /STANDARD=PORTABLE qualifier on the CC command line.

User Action: No action is necessary if you do not require compatibility with other C compilers. Otherwise, correct the line.

NONPORTCOMP, Comparisons between pointers and integers may not be portable.

NONPORTCOMP, Comparisons between pointers to different types may not be portable.

Informational: You compared the value of a pointer or an address expression with either an integer expression, a nonzero integer constant, or a pointer or address expression of a different type. Such usage may not be portable and is not recommended. The only portable pointer comparisons are between a pointer and the integer constant 0, or between pointers to objects of the same type. This message is issued only if you specified `/STANDARD=PORTABLE` on the CC command line.

User Action: Cast one of the operands to be the same type as the other.

NONPORTCONST, Character constant `****` may not be portable.

Warning: VAX C allows up to four characters to be specified in a character constant, but other compilers may not. The compiler issues this message if you use the `/STANDARD=PORTABLE` qualifier on the CC command line.

User Action: If you do not require portability, no action is necessary.

NONPORTCVT, Conversions between pointers and integers may not be portable.

NONPORTCVT, Conversions between pointers to different types may not be portable.

Informational: You converted a pointer or an address expression to an integer type or to a different pointer type, or an integer expression or a nonzero integer constant to a pointer type. Such usage may not be portable and is not recommended. The only portable assignments are between pointers to objects of the same type or conversion of the integer constant 0 to any pointer type. This message is issued only if you specified `/STANDARD=PORTABLE` on the CC command line.

User Action: Use an explicit cast to perform the conversion.

NONPORTINCLUDE, #include of a library module is not portable.

Informational: The specification of a library module name in an **#include** preprocessor directive is VAX C specific and is not portable. This message is issued if you specified the /STANDARD=PORTABLE qualifier on the CC command line.

User Action: No action is necessary if you do not require compatibility with other C compilers.

NONPORTINIT, Automatic initialization for "*****" may not be portable.

Informational: You initialized an array or structure of storage class **auto**. This message is issued if you specified /STANDARD=PORTABLE on the CC command line.

User Action: If you require portability, use separate assignment statement(s) to set the initial value(s).

NONPORTOPTION, The "*****" function definition option is not portable.

Informational: The VAX C function definition options are VAX C specific and are not portable. The compiler issued this message if you used /STANDARD=PORTABLE on the CC command line.

User Action: No action is necessary if you do not require compatibility with other C compilers.

NONPORTPPDIRX, The **** preprocessor directive is not portable.

Informational: You used the **#dictionary** or **#module** preprocessor directive.

These directives are VAX C specific and may not be recognized by other compilers. The compiler issues this message if you specified /STANDARD=PORTABLE on the CC command line.

User Action: No action is necessary if you do not require program portability.

NONPORTPTR, The use of an integer value as a pointer qualifier for "*****" may not be portable.

Informational: In a reference to a structure or union member accessed by the "->" operator, the qualifying expression to the left of the "->" should have a pointer value. VAX C allows the use of integer values as well, but such usage is not portable. This

message is issued if you specify /STANDARD=PORTABLE on the CC command line.

User Action: Either use a true pointer expression as the qualifier or cast the integer expression as an appropriate structure or union pointer.

NONPORTTYPE, Data type "****" is not portable.

Informational: You used either of the data types **variant_struct** or **variant_union**, which are VAX C specific. This message is issued if you specify /STANDARD=PORTABLE on the CC command line.

User Action: No action is necessary if you do not require program portability.

NONSEQUITUR, "****" is not a member of the specified structure or union.

Informational: In a reference to the indicated member name, you specified a qualifier that does not represent the structure or union to which the member belonged.

The reference is valid, because the member name is unique and the offset can be resolved unambiguously. This use of member names is maintained only for compatibility with older programs.

User Action: If the qualifier is a pointer, cast it as a pointer to the appropriate structure or union.

NONTERMCHAR, Nonterminated character constant; **** assumed.

Warning: The compiler encountered the end of the source line before the end of a character constant. The compiler assumed the indicated value.

User Action: Correct the constant.

NONTERMNULCHAR, Nonterminated character constant contains no characters; '\0' assumed.

Warning: The compiler detected a single apostrophe (') at the end of the source line.

User Action: Check to see if there is an extra apostrophe; otherwise, correct the constant.

NONTERMSTRING, Nonterminated string constant; quotes added at end of line.

Warning: The compiler encountered the end of the source line before the end of a character string. The compiler inserted a quotation mark (") at the end of the line.

User Action: Check to see if the string should be continued on the following line; if so, insert a backslash (\) at the end of the line. Otherwise, check for the missing quotation mark.

NOOPTIMIZATION, Complex control flow caused optimization to be suppressed for procedure or function "*****".

Informational: Optimization was not performed for the indicated function.

User Action: To take advantage of optimization, simplify the control flow within the indicated function.

NOSUBSTITUTION, Macro substitution cannot be performed during the scan of a macro reference; "*****" not substituted; directive ignored.

NOSUBSTITUTION, Macro substitution cannot be performed during the scan of a macro reference; "*****" not substituted; true expression assumed.

Warning: You wrote a complex macro reference that contained a preprocessor directive, which in turn contained another macro reference. For example:

```
macref1 ( arg1,  
#include MACREF2  
.  
.  
.  
, argn)
```

The substitution of MACREF2 was not performed and the directive containing it was ignored. If the directive was **#if** or **#elif**, the expression would be assumed to be "true."

User Action: Restructure your code so that the directive is not contained within the macro reference.

NOTFUNCTION, Function-valued expression not found.

Error: You used an expression in the context of a function call, but the expression does not evaluate to a function.

User Action: Make sure that the expression properly evaluates to a function; also make sure that you properly dereference any pointer to a function.

NOTPARAMETER, "****" is not listed in the function's formal parameter list; treated as if declared internally.

Warning: You declared the specified identifier as a function parameter, but the identifier does not appear in the parameter list. For example:

```
f(a) int a,b; { . . . }
```

The identifier `b` does not appear in function `f`'s formal parameter list. Its declaration is not portable, and is probably a coding error. The compiler treats `b` as if it were declared inside the function definition; in this case, `b` becomes an automatic variable.

User Action: Correct the declaration or the parameter list.

NOTPOINTER, Address-valued expression not found.

Error: You used an expression in a context requiring a pointer value, but the expression did not evaluate to an address.

User Action: Make sure that the expression evaluates to a pointer value.

NOTSAFECALL, Function "****" inhibited loop decomposition at loop control variable "****"

Informational: You have a function that inhibited loop decomposition at a loop control variable. This message is issued if you specified the `/PARALLEL` qualifier on the `CC` command line.

User Action: If the loop must be decomposed and if the function does not introduce a dependency, insert a `safe_call` pragma.

NOTSWITCH, Default labels and case labels are valid only in "switch" statements.

Error: You used `case` or `default` as a label outside the body of a `switch` statement.

User Action: Check for unmatched braces that may have prematurely terminated the most recent `switch` statement.

NOTUNIQUE, "****" is not a unique member name in this context.

Error: You used the same member name in more than one structure or union definition, and then used that member name as an offset from some other structure or union. Since the compiler had no way of knowing which member definition to use as an offset, a message was generated.

User Action: To avoid ambiguities, try to make all member names unique.

NULCHARCON, Character constant contains no characters; '\0' assumed.

Warning: You used '' for an ASCII NUL character instead of '\0'.

User Action: Use '\0'.

NULHEXCON, Hexadecimal constant contains no digits; 0X0 assumed.

Warning: You specified a constant such as 0X or 0x.

User Action: Be sure that 0 is a valid value in this context; if so, change the constant to 0x0.

OVERDRAFT, **** has gone into DISK QUOTA overdraft.

Informational: Your disk I/O quota was exceeded while writing to a file.

User Action: Purge your directories to create more space or increase your disk I/O quota.

PARAMNOTUSED, Macro parameter "*****" is not referenced in the definition.

Informational: A macro definition had more parameters than appeared in its substitution. For example:

```
#define m(a,b,c) a*b
```

User Action: Specify the extra parameter in the substitution or, if it is superfluous, delete it from the parameter list. (This is a possible programming error.)

PARAMREDECL, This declaration of "****" overrides a formal parameter.

Warning: Your source program contained a redeclaration of one of the function's formal parameters. For example:

```
f(a) { int a; }
```

You cannot reference the parameter from within the function.

User Action: If the declaration is misplaced, move it to a position between the function header and the left brace at the beginning of the function body. Otherwise, rename one of the identifiers.

PARSTKOVRFW, Parse stack overflow.

Fatal: The source code in your program was too complex, containing statements nested too deeply.

User Action: Simplify the program.

PPUNEXPEOF, Unexpected end-of-file encountered in preprocessor directive; directive ignored.

Warning: The compiler detected the end of the source file while trying to read a continuation of a preprocessor directive.

User Action: Check for nonterminated comments, character strings, and other constructs that can span several lines of code.

PRAGMASYNTAX, Syntax error in #pragma preprocessor directive; directive ignored.

Warning: You have incorrectly coded the directive.

User Action: Correct the error. Check for misspellings or punctuation errors.

PTRDEPENDENCE, Pointer "****" inhibited loop decomposition at loop control variable "****"

Informational: You have a pointer that inhibited loop decomposition at a loop control variable. This message is issued if you specified the /PARALLEL qualifier on the CC command line.

User Action: If the loop must be decomposed and if the pointer does not introduce a dependency, use the **ignore_dependency** decomposition pragma.

PTRFLOATCVT, Operand of pointer addition or subtraction converted from floating-point to integer.

Warning: You combined a pointer operand with a floating-point value. For example:

```
int i,*ip;
.
.
.
i = ip + 2.;
```

User Action: Make sure that pointers are used only with other pointers or with integers. In the previous example and in similar situations, remove the decimal point from the literal constant.

QUALNOTLVALUE, The qualifier for "****" is not a valid lvalue.

Error: In a reference to a structure or union member accessed by the period operator (.), the qualifying expression to the left of the period must be an lvalue.

User Action: Correct the qualifying expression.

QUALNOTSTRUCT, The qualifier for "****" is not a structure or union.

Informational: In a reference to a structure or union member, the qualifying expression to the left of the period operator (.) or right-arrow operator (->) did not represent a structure or union.

User Action: Check for possible spelling errors.

REDEFPROTO, This prototype conflicts with either the function definition or with a function prototype which appears earlier in the file.

Warning: The prototype conflicts with a previous declaration of this function, either in number, type of arguments, or in the return type of the function.

User Action: Determine what attribute does not match and what the correct attribute should be. Correct the invalid definition.

REDUNDANT, The operand of the "&" operator is already an address.
The "&" is ignored.

Informational: You specified & in front of an array or function name. The message is issued if you specified /STANDARD=PORTABLE on the CC command line.

User Action: Make sure that you intend to pass the address of the array or function. If you require portability, remove the redundant &.

REGADDR, Taking the address of register variable "****" is not portable and causes its storage class to be changed to auto.

Informational: You used the unary ampersand operator (&) to take the address of a register variable. VAX C changes the storage class of the variable from **register** to **auto**. This allows the address of the variable to be taken. The message is used if you specified the /STANDARD=PORTABLE qualifier on the CC command line.

User Action: No user action is needed if you do not require compatibility with other C compilers. If you do require compatibility, change the storage class of the variable from **register** to **auto**.

REPABBREV, Replaced abbreviation **** with ****.

Warning: You abbreviated a reserved word.

User Action: Complete the spelling of all reserved words.

REPLACED, Replaced **** with ****.

Warning: The compiler replaced an invalid macro with a different macro. (Programs that contain syntax errors usually generate this message.)

User Action: Check for incorrect syntax.

REPOVERFLOW, Length of replacement text exceeds maximum buffer capacity; "****" not substituted.

Error: The length of the replacement text for a macro reference or the length of the text plus the rest of the line exceeded the implementation's limit.

User Action: Shorten the replacement text or use multiple substitutions to achieve the desired result.

RESERVED, "****" is a reserved identifier; directive ignored.

Warning: You have specified a reserved identifier name in a **#define** or **#undef** preprocessor directive. The following reserved names may not be redefined or undefined:

- `__DATE__`
- `__FILE__`
- `defined`
- `__TIME__`
- `__LINE__`

User Action: Choose a different spelling for the identifier.

SCALEFACTOR, The CDD description for "****" specifies a scale factor of **** ; the scale factor is being ignored.

Informational: VAX C does not support scaled arithmetic.

User Action: Make sure that you appropriately scale computations involving this item.

SEMICOLONADDED, Semicolon added at the end of the previous source line.

Warning: A missing semicolon was added to the line prior to the line numbered in this message.

User Action: Check the previous line carefully and add the semicolon in the appropriate place.

SUMMARY, Completed with **** errors, **** suppressed warning(s), and **** informational messages.

SUMMARY, Completed with **** errors, **** warnings, and **** information messages

Informational: This message indicates the number of compiler messages (errors, warnings, and informationals) issued during the compilation process. You can suppress informational and warning messages using the `/[NO]WARNINGS` CC command-line qualifier (see Chapter 1).

User Action: Consider the individual messages and recompile if necessary.

SYMTABOVFL, The total number of symbol table pages exceeds the implementation's limit.

Fatal: The program was too complex.

User Action: Simplify the program by reducing the number and size of variables and other names, constants, and so forth.

SYNTAXERROR, **** Found **** when expecting ****.

Error: The syntax error shown prevented the generation of an object file.

User Action: Correct the errors shown.

TBLOVRFLW, Internal table overflow, too many procedures, external symbols (psects), or the program is too complex.

Fatal: Either the source file contains too many functions or expressions, or the compiler has overflowed its virtual address space.

User Action: Reduce the size of the source file by dividing it into smaller, separate files, or change the logic of the program to reduce the number of complicated expressions.

TOOFEWMACARGS, Argument list for macro "****" contains too few arguments; missing arguments assumed to be null.

Warning: You wrote a reference to the indicated macro with fewer arguments than were specified in its definition.

User Action: Make sure that the number of arguments in the macro reference is the same as the number of parameters in the definition.

TOOFEWPRAGIDS, At least two identifiers must be specified in this #pragma preprocessor directive; directive ignored.

Warning: You have coded only one identifier in the list of identifiers in this directive; at least two are required. The directive was ignored by the compiler.

User Action: Remove the directive or add the missing identifiers, as appropriate.

TOOMANYCHAR, Character constant contains too many characters; truncated to ****.

Warning: The length of a character constant exceeded the implementation limit (four characters). The constant was truncated to the indicated value.

User Action: Reduce the length of the indicated character constant to four or fewer characters.

TOOMANYERR, The total number of errors exceeds the limit of 100.

Fatal: The compiler reported more than 100 error messages in this compilation. The compilation ended at this point.

User Action: Correct the errors reported in previous compiler messages and recompile the program.

TOOMANYFUNARGS, Function reference specifies too many arguments; excess arguments ignored.

Warning: You called a function with more than 253 arguments. The compiler passed only the first 253 arguments; the compiler ignored the remainder.

User Action: Shorten the argument list.

TOOMANYINITS, The initializer list for "****" specifies too many initializers; excess initializers ignored.

Warning: You specified too many initializers for the indicated variable. (If the indicated item is an array or structure, it may be only partially initialized.)

User Action: Make sure that all braces near the initializer sublists are balanced; if the item being initialized is or contains an array, make sure that you account for all dimensions.

TOOMANYMACARGS, Argument list for macro "****" contains too many arguments; excess arguments ignored.

Warning: You wrote a reference to the indicated macro with more arguments than were specified in its definition.

User Action: Make sure that the number of arguments in the macro reference is the same as the number of parameters in the definition.

TOOMANYMACPARM, Parameter list for macro "****" contains too many parameters; excess parameters ignored.

Warning: The number of macro parameters in a **#define** preprocessor directive exceeded the implementation limit of 64.

User Action: Rewrite the macro definition so that it uses 64 or fewer parameters.

TOOMANYSTR, String constant contains too many characters; truncated.

Warning: You wrote a character-string constant whose length exceeded the implementation's limit of 65,535 characters.

User Action: Shorten the string.

TRUNCFLOAT, Double-precision floating-point constant cannot be converted to single precision; 0.0 assumed.

Warning: You specified a double-precision constant in an expression involving a conversion to single precision, but the constant's value was too small to be represented in single precision.

User Action: Ensure that 0 is a valid value in this context; if necessary, redeclare the conversion target as **double**.

TRUNCSTRINIT, String initializer for "****" contains too many characters to fit; truncated.

Warning: If the variable was a simple one-dimensional array, the initializer was truncated (so that the length of the initializer was array-1) and the null byte was added to the end of the array. If the array is a multidimensional array or an array within a structure, the initializer was truncated to the length of the array and a null byte was not added.

User Action: Treat arrays of characters as strings allowing for the null byte at the end of the array. The special case of multidimensional arrays and arrays within structures should be taken into account, especially when you do not want the null byte to be truncated.

TYPECONFLICT, "****" conflicts with a previous data type in this declaration; previous data type ignored.

Warning: You specified more than one data-type specifier in this declaration, and the indicated specifier conflicted with a previous one.

User Action: Check for a missing semicolon in the previous declaration; otherwise, make sure that all specifiers are compatible.

TYPEINLIST, The type of "****" was specified in the parameter list. This declaration is ignored.

Warning: The function definition uses the prototype format but still contains a declaration of this parameter in the parameter declaration section.

User Action: Eliminate the redundant declaration.

UABORT, Compilation terminated by user.

Fatal: The compilation was terminated by a DCL CTRL/C command.

User Action: None.

UNDECLARED, "****" is not declared within the scope of this usage.

Error: You referred to an undeclared variable. (You must declare variables before you use them.)

User Action: Check the spelling of the identifier, or add a declaration for it, if appropriate.

UNDECLARED, "****" is not declared prior to this #pragma preprocessor directive; directive ignored.

Warning: This directive lists an identifier that has not yet been declared. The entire directive has been ignored by the compiler.

User Action: Check the spelling of the identifier or add a declaration for it, if appropriate.

UNDEFIFMAC, "****" is not a currently defined macro; constant zero assumed.

Informational: The identifier in a constant expression in an **#if** or **#elif** preprocessor directive was not currently defined as a macro. The expression was evaluated as if the identifier were a constant zero. This message is only generated if you compile with **/STANDARD=PORTABLE**.

User Action: Define the identifier as a macro or remove the reference to it.

UNDEFLABEL, Label "****" is undefined in this function.

Error: You wrote "**goto** label-name" for an undefined label. The scope of a label name is restricted to the function in which it is used as a label; **goto** statements cannot branch to labels inside other functions.

User Action: Check the spelling of the label name or make other corrections as appropriate.

UNDEFMACRO, "****" is already undefined; directive ignored.

Warning: The specified identifier (in an **#undef** directive) was either never defined or else occurred in a previous **#undef**.

User Action: Remove the **#undef**, or, if applicable, add the correct definition of the identifier.

UNDEFSTRUCT, "****" is a structure or union type that is not fully defined at this point in the compilation.

Error: You used a name in the context of a structure or union tag, but the name is either undefined or is not yet fully defined as a tag.

User Action: Check the spelling of the name, and make sure that it is fully defined as a tag before using it.

UNEXPEND, Unexpected end-of-**** encountered in "****" preprocessor directive; directive ignored.

Warning: The end of the directive or end of the source file was encountered before the directive was completely processed.

User Action: Check for an incomplete comment within the definition, or check for a missing continuation of the directive.

UNEXPEOF, Unexpected end-of-file encountered in a ****.

Error: The compiler encountered the end of the source file while scanning for the end of a string constant or a comment.

User Action: Make sure that string constants and comments are properly terminated.

UNEXPPDIRX, Unexpected **** preprocessor directive encountered; directive ignored.

Warning: The specified directive occurred out of place and was ignored.

User Action: Check the logic of all directives in the program to be sure that it is valid.

UNKSIZEOF, Operand of sizeof has an unknown size; 0 assumed.

Warning: The operand of a **sizeof** operator was an array whose size was unknown at compile time. A size of 0 was assumed.

User Action: Change the declaration of the array to specify the appropriate dimension bound.

UNRECCHAR, Unrecognized character ignored.

Warning: The line contained either an entirely meaningless character or one that appears out of its proper context; for example, a number sign (#) that was not the first character on a line.

User Action: Move or remove the character.

UNRECPRAGMA, Unrecognized #pragma preprocessor directive ignored.

Informational: You have specified a **#pragma** preprocessor directive that is not recognized by VAX C.

User Action: Correct the syntactic or semantic error that rendered the directive unrecognizable. Common errors include misspelled parameters.

UNSUPPORTEDLCV, Loop decomposition inhibited due to unsupported loop control variable.

Informational: A function was expanded inline and its formal parameter was used as a loop-control variable.

User Action: Declare the loop-control variable to be the automatic storage class.

UNSUPPORTEDOP, Variable "****" has an unsupported type which inhibited loop decomposition at loop control variable "****".

Informational: A variable that is modified in the loop is of a type not currently supported by VAX C decomposition processing. This message is issued if you specified the /PARALLEL qualifier on the CC command line.

User Action: No action.

UNSUPPTYPE, The CDD description for "*****" specifies a data type not supported in C.

Informational: The compiler could not represent the indicated item in a VAX C construct. The compiler generated a declaration of a structure whose length was the same as the length of the unsupported data type.

User Action: Change the CDD description to specify a supported data type, if you require a precise representation in VAX C.

VARNOTMEMBER, A variant aggregate must be a member of a struct or union.

Error: You tried to specify a **variant_struct** or a **variant_union** outside of an aggregate declaration.

User Action: If you intend to use the structure or union as declared, and if the structure or union is the outermost aggregate in a group of nested aggregates, replace the variant keywords with **struct** or **union**. If you intend to use the structure or union as a variant aggregate, and if the structure or union is otherwise properly declared, nest the declaration within a valid structure or union declaration. If you use the **variant_struct** or **variant_union** keywords in declarations other than structure or union declarations, remove the variant keywords.

VOIDCALL, A "void" function cannot be invoked in a context where a value is expected.

Error: You coded a call to a function declared as **void**, but the call appeared in a context where a return value was expected.

User Action: Move the function call to a different context, or if the function does return a value, declare it to be **void**.

VOIDEXPR, A "void" expression cannot be used in a context where a value is expected.

Error: You cast an expression to be **void**, but the expression was used in a context where its value was required.

User Action: Remove the cast, or move the expression to a context that requires no return value.

VOIDNOTFUNC, "****" is not declared to be a function; only functions may be declared "void".

Error: You declared an object other than a function to be **void**.

User Action: Check the syntax of the declarator. You may find the output produced by the /SHOW=SYMBOLS CC command-line qualifier to be helpful in diagnosing this problem.

VOIDRETURN, A "return" statement in a "void" function may not specify a value; expression ignored.

Warning: You specified a value in a **return** statement within a function declared as **void**.

User Action: Either remove the return value or redefine the function as returning the appropriate data type.

Optional Programming/ Productivity Tools

This appendix provides an overview of optional programming productivity tools. These tools are not included with the VAX C software; they must be purchased separately. Using these tools can increase your productivity as a VAX C programmer. Contact your DIGITAL sales representative for more information about these tools.

This appendix discusses the following optional productivity products:

- The VAX Language-Sensitive Editor—LSE (Section C.1)
- The VAX Source Code Analyzer—SCA (Section C.2)

C.1 Using VAX LSE with VAX C

The VAX Language-Sensitive Editor (LSE) is a powerful and flexible text editor designed specifically for software development. LSE has important features that help you produce syntactically correct code in VAX C.

To invoke LSE, specify the LSEEDIT command followed by a file name with a C file type at the DCL prompt. For example:

```
$ LSEEDIT USER.C
```

The following sections describe some of the key features of LSE. Section C.1.1 discusses how to enter source code using LSE. Section C.1.2 describes LSE's compiler interface features. Section C.1.3 gives examples of how to generate VAX C source code with LSE.

For more details on the advanced features of LSE and SCA, see the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer*.

C.1.1 Entering Source Code Using Tokens and Placeholders

LSE's language-sensitive features simplify the tasks of developing and maintaining software systems. These features include language-specific placeholders and tokens, aliases, comment and indentation control, and templates for subroutine libraries.

You can use LSE as a traditional text editor. In addition, you can use the power of LSE's tokens and placeholders to step through each program construct and supply text for those constructs that need it.

Placeholders are markers in the source code indicating where you can provide program text. These placeholders help you to supply the appropriate syntax in a given context. You do not need to type placeholders; they are inserted for you by LSE. Placeholders are surrounded by brackets or braces and at (@) signs.

Table C-1 describes the types of LSE placeholders.

Table C-1: LSE Placeholders

Type of Placeholder	Description
Terminal	Provides text strings that describe valid replacements for the placeholder.
Nonterminal	Expands into additional language constructs.
Menu	Provides a list of options corresponding to the placeholder.

Placeholders are either optional or required. Required placeholders, indicated by braces ({}), represent places in the source code where you must provide program text. Optional placeholders, indicated by brackets ([]), represent places in the source code where you can either provide additional constructs or erase the placeholder.

You can move forward or backward from placeholder to placeholder. In addition, you can delete or expand placeholders as needed.

Tokens typically represent keywords in VAX C. When expanded, tokens provide additional language constructs. You can type tokens directly into the buffer. You use tokens in situations, such as modifying an existing program, where you want to add additional language constructs and there are no placeholders. For example, typing IF and entering the EXPAND command causes a template for an IF construct to appear on your screen. You can also use tokens to bypass long menus in situations where expanding a placeholder, such as {@statement@}, will result in a lengthy menu.

You can use tokens to insert text when editing an existing file by typing the name for a function or keyword and entering the EXPAND command.

LSE provides commands that allow you to manipulate tokens and placeholders. Table C-2 shows these commands and their default key bindings.

Table C-2: Commands to Manipulate Tokens and Placeholders

Command	Key Binding	Function
EXPAND	CTRL/E	Expands a placeholder.
UNEXPAND	PF1-CTRL/E	Reverses the effect of the most recent placeholder expansion.
GOTO PLACEHOLDER/FORWARD	CTRL/N	Moves the cursor forward to the next placeholder.
GOTO PLACEHOLDER/REVERSE	CTRL/P	Moves the cursor backward to the next placeholder.
ERASE PLACEHOLDER/FORWARD	CTRL/K	Erases a placeholder.
UNERASE PLACEHOLDER	PF1-CTRL/K	Restores the most recently erased placeholder.
↓	Down-arrow	Moves the indicator downward through a screen menu.
↑	Up-arrow	Moves the indicator upward through a screen menu.
<input type="text" value="ENTER"/> <input type="text" value="RETURN"/>	{ ENTER } { RETURN }	Selects a menu option.

To display a list of all the defined tokens provided by VAX C, enter the LSE command SHOW TOKEN as follows:

```
LSE> SHOW TOKEN
```

To display a list of all the defined placeholders provided by VAX C, enter the LSE command SHOW PLACEHOLDER as follows:

```
LSE> SHOW PLACEHOLDER
```

To put either list into a separate file, first enter the appropriate **SHOW** command to put the list into the **\$\$SHOW** buffer. Then enter the following commands:

```
LSE> GOTO BUFFER $$SHOW  
LSE> WRITE filename
```

To obtain a hard copy of the list, use the **PRINT** command at **DCL** level to print the file you created.

To obtain information about a particular token or placeholder, you can also specify a token name or placeholder name after the **SHOW TOKEN** or **SHOW PLACEHOLDER** command.

C.1.2 Compiling Source Code

To compile your source code and to review compilation errors without leaving the editing session, use the **LSE** commands **COMPILE** and **REVIEW**. The **COMPILE** command issues a **DCL** command in a subprocess to invoke the **VAX C** compiler. The compiler then generates a file of compile-time diagnostic information that **LSE** can use to review compilation errors. The diagnostic information is generated with the **/DIAGNOSTICS** qualifier that **LSE** appends to the compilation command.

For example, if you enter the **COMPILE** command while in the buffer **USER.C**, the resulting **DCL** command is as follows:

```
$ CC USER.C/DIAGNOSTICS=USER.DIA
```

LSE supports all the **VAX C** compiler's command qualifiers as well as user-supplied command procedures. You can specify **DCL** qualifiers, such as the **/LIBRARY** qualifier, when invoking the compiler from **LSE**.

The **REVIEW** command displays any diagnostic messages that result from a compilation. **LSE** displays the compilation errors in one window and the corresponding source code in a second window. This multiwindow capability allows you to review your errors while examining the associated source code. This capability eliminates tedious steps in the error-correction process, and helps ensure that all errors are fixed before you compile your program again.

LSE provides several commands to help you review errors and examine your source code. Table C-3 lists these commands and their default key bindings where applicable.

Table C-3: LSE Commands to Review and Examine Source Code

Command	Key Binding	Function
COMPILE	None	Compiles the contents of the source buffer.
COMPILE/REVIEW	None	Compiles the contents of the source buffer, puts LSE into REVIEW mode, and displays any errors resulting from the compilation.
REVIEW	None	Performs the same function as the /REVIEW qualifier on the COMPILE command: puts LSE into REVIEW mode, and displays any errors resulting from the last compilation.
END REVIEW	None	Removes the buffer \$REVIEW from the screen; returns the cursor to a single window containing the source buffer.
GOTO SOURCE	CTRL/G	Moves the cursor to the source buffer that contains the error.
NEXT STEP	CTRL/F	Moves the cursor to the next error in the buffer \$REVIEW.
PREVIOUS STEP	CTRL/B	Moves the cursor to the previous error in the buffer \$REVIEW.
↓ ↑	{ Down arrow } { Up arrow }	Moves the cursor within a buffer.

C.1.2.1 Pragma Insertions and Decomposition

LSE can aid in the creation of programs that use the parallel-processing capability of VAX C. If you use both the /PARALLEL and /DIAGNOSTICS compilation qualifiers, LSE provides informational messages during the REVIEW period about the ability of the compiler to decompose loops.

During the REVIEW period, you can place one of the following pragmas into your code that alters how the compiler decomposes certain loops and can eliminate diagnostic messages:

- **#pragma ignore_dependency**
- **#pragma safe_call**
- **#pragma sequential_loop**

See Section 3.7 for more information on using these directives.

C.1.3 Examples

This section describes the special features of VAX C available through LSE and provides examples of VAX C code written with LSE.

The following examples show the expansions of some common VAX C tokens and placeholders. The examples are expanded to show the formats and guidelines LSE provides; however, not all of the examples are fully expanded.

The examples show expansions for the following VAX C features:

- Preprocessor lines
- External definitions
- Function definitions
- Block declarations
- Statements and expressions

Instructions and explanations precede each example, and an arrow (→) indicates where in the code an action occurred.

To invoke LSE and the VAX C language, use the following syntax:

```
LSEDIT [qualifier . . . ] filename.C
```

Table C-2 lists the commands that manipulate tokens and placeholders.

When you use the editor to create a new VAX C program, the initial string {`@compilation_unit@`} appears at the top of the screen. Expanding the initial string produces the following:

```
->      [ @#module@ ]
        [ @module_level_comments@ ]
        [ @include_files@ ]
        [ @macro_definitions@ ]
        [ @preprocessor_line@ ] . . .
        [ @comment@ ] . . .
        [ @external_definition@ ] . . .
        [ @function_definition@ ] . . .
```

C.1.3.1 Preprocessor Lines

Erase the `[@#module@]`, `[@module_level_comments@]`, `[@include_files@]`, and `[@macro_definitions@]`. The cursor is then positioned on `[@preprocessor_line@]`. Expand `[@preprocessor_line@]` to duplicate it and display a menu. Select the option `#include` as follows:

```
->      #include
        [preprocessor_line@] . . .
        [comment@] . . .
        [external_definition@] . . .
        [function_definition@] . . .
```

After selecting the `#include` option, another menu appears that lists the types of `#include` statements. Select the option `#include {include_module_name@}`.

```
->      #include      {include_module_name@}
        [preprocessor_line@] . . .
        [comment@] . . .
        [external_definition@] . . .
        [function_definition@] . . .
```

Type the value `stdio` over the placeholder `{include_module_name@}`.

C.1.3.2 External Definition

```
[preprocessor_line@] . . .
[comment@] . . .
[external_definition@] . . .
[function_definition@] . . .
```

Erase the placeholders `[@preprocessor_line@]` and `[@comment@]`. Expand the placeholder `[@external_definition@]` to display a menu and select the option `static` as follows:

```
->      static [readonly@] [data_modifiers@] . . . [data_type@] {init_declarator@} . . . ;
        [external_definition@] . . .
        [function_definition@] . . .
```

Erase the placeholder `[@readonly@]` and `[@data_modifiers@]` and type the value `double` over the placeholder `[@data_type@]` as follows:

```
->      static double      {init_declarator@} . . . ;
        [external_definition@] . . .
        [function_definition@] . . .
```

Expand `{@init_declarator@}` to produce the following:

```
-> static double    {@declarator@} [=@= initializer@], {@init_declarator@} ... ;
    [external_definition@] ...
    [function_definition@] ...
```

Erase the duplicated list placeholder `{@init_declarator@} ...` (the separator text `;` will appear immediately after the inserted text). Expand the placeholder `{@declarator@}` to display a menu and select the option `{@identifier@}`; type the value number over `{@identifier@}` as follows:

```
-> static double    number [=@= initializer@];
    [external_definition@] ...
    [function_definition@] ...
```

Expand the placeholder `[=@= initializer@]` to display a menu and select the option `= {@init_constant_expression@}` as follows:

```
-> static double    number = {@init_constant_expression@};
    [external_definition@] ...
    [function_definition@] ...
```

Type the value `30.0` over `{@init_constant_expression@}` as follows:

```
-> static double    number = 30.0;
    [external_definition@] ...
    [function_definition@] ...
```

C.1.3.3 Function Definition

```
[external_definition@] ...
[function_definition@] ...
```

Erase the placeholders `[@preprocessor_line@]`, `[@comment@]`, and `[@external_definition@]`. Expand `[@function_definition@]` to display a menu and select the option `{@function_def@}` as follows:

```
-> [function_level_comments@]
    [static@]    [data_type@]    {@function_name@} ([parameter@] ... )
    [param_decl@] ...
    {
        [block_decl@] ...
        [statement@] ...
    }
    [function_definition@] ...
```

Because `[@function_definition@]` is a list placeholder, a copy of it is placed after the body of `{@function_def@}`. Since `[@function_definition@]` is optional, for purposes of this example erase it.

Erase the placeholders `[@function_level_comments@]` and `[@static@]`.
Expand `[@data_type@]` to display a menu and select the option
`[@unsigned@]#int` as follows:

```
->      [@unsigned@] int      {@function_name@} ([@parameter@] ... )
        [@param_decl@] ...
        {
            [@block_decl@] ...
            {@statement@} ...
        }
```

Erase `[@unsigned@]` and type the value `get_string` over `{@function_name@}`.
Expand the placeholder `[@parameter@]` to produce the following:

```
->      int      get_string ({@identifier@}, [@parameter@] ... )
        [@param_decl@] ...
        {
            [@block_decl@] ...
            {@statement@} ...
        }
```

Type the value `string` over `{@identifier@}`, and expand `[@parameter@]` again
to produce the following:

```
->      int      get_string (string, {@identifier@}, [@parameter@] ... )
        [@param_decl@] ...
        {
            [@block_decl@] ...
            {@statement@} ...
        }
```

Type the value `limit` over `{@identifier@}` and erase `[@parameter@] ...` as
follows:

```
->      int      get_string (string, limit)
        [@param_decl@] ...
        {
            [@block_decl@] ...
            {@statement@} ...
        }
```

Expand the placeholder `[@param_decl@]` as follows:

```
->      int      get_string (string, limit)
        [@register@]  [@data_modifiers@] ...  [@data_type@] {@declarator@} ... ;
        [@param_decl@] ...
        {
            [@block_decl@] ...
            {@statement@} ...
        }
```

Expand the placeholder `[@register@]` to produce the value register. Erase the placeholder `[@data_modifiers@]`. Type the value `char` over the placeholder `[@data_type@]` as follows:

```
int    get_string (string, limit)
-> register char    {@declarator@} . . . ;
    {@param_decl@} . . .
    {
        {@block_decl@} . . .
        {@statement@} . . .
    }
```

Expand `{@declarator@}` to display a menu and select the array format `[@declarator@] [[@constant_expression@]]`. Erase the duplicated list placeholder `{@declarator@}` to produce the following:

```
int    get_string (string, limit)
-> register char    {@declarator@} [[@constant_expression@]];
    {@param_decl@} . . .
    {
        {@block_decl@} . . .
        {@statement@} . . .
    }
```

Type the value `string` over `{@declarator@}` and erase `[@constant_expression@]` as follows:

```
int    get_string (string, limit)
-> register char    string [];
    {@param_decl@} . . .
    {
        {@block_decl@} . . .
        {@statement@} . . .
    }
```

Expand `[@param_decl@]` again and erase the placeholders `[@register@]` and `[@data_modifiers@]` as follows:

```
int    get_string (string, limit)
-> register char    string [];
    {[@data_type@] {@declarator@} . . . ;
    {@param_decl@} . . .
    {
        {@block_decl@} . . .
        {@statement@} . . .
    }
```

Type the value `int` over `[@data_type@]` and expand `{@declarator@}` to display a menu. Select the option `{@identifier@}` as follows:

```

        int    get_string (string, limit)
        register char    string [];
->    int    {@identifier@}, {@declarator@} ... ;
        {@param_decl@} ...
        {
            {@block_decl@} ...
            {@statement@} ...
        }

```

Type the value limit over {@identifier@} and erase {@declarator@} ... and {@param_decl@} ... as follows:

```

        int    get_string (string, limit)
        register char    string [];
->    int    limit;
        {
            {@block_decl@} ...
            {@statement@} ...
        }

```

C.1.3.4 Block Declaration

```

[function_definition@] ...

```

Expand the placeholder [function_definition@] to display a menu and select the option main_function_def as follows:

```

->    [function_level_comments@]
    {@main () OR main function that accepts arguments from the command line@}
    {
        {@block_decl@} ...
        {@statement@} ...
    }
[function_definition@] ...

```

Erase the placeholder [function_level_comments@] and the duplicated list placeholder [function_definition@]. Expand {@main () OR main function that accepts arguments from the command line@} to display a menu and select the option main () as follows:

```

->    main ()
        {
            {@block_decl@} ...
            {@statement@} ...
        }

```

Expand the placeholder [block_decl@] to display a menu and select the option [data_modifiers@] [data_type@] [init_declarator@] as follows:

```

main ()
{
->   [@data_modifiers@]   [@data_type@]   [@init_declarator@] ... ;
   [@block_decl@] ...
   {@statement@} ...
}

```

Erase the placeholder `[@data_modifiers@]` and expand `[@data_type@]` to display a menu. Expand option `struct` automatically expands to a menu, from which you select the option `{@struct struct_decl@}` as follows:

```

main ()
{
->   {@struct struct_decl@}   [@init_declarator@] ... ;
   [@block_decl@] ...
   {@statement@} ...
}

```

Expand `{@struct struct_decl@}` to produce the following:

```

main ()
{
->   struct
   {
       {@member_decl@} ...
   }   [@init_declarator@] ... ;
   [@block_decl@] ...
   {@statement@} ...
}

```

Expand `{@member_decl@}` to display a menu and select the option `[@data_type@] {@declarator@} ... ;` as follows:

```

main ()
{
   struct
   {
->       [@data_type@]   {@declarator@} ... ;
       [@member_decl@] ...
   }   [@init_declarator@] ... ;
   [@block_decl@] ...
   {@statement@} ...
}

```

Type the value `char` over `[@data_type@]`. Expand `{@declarator@}` to display a menu and select the option `{@declarator@} [[@constant_expression@]]`. Erase the duplicated list placeholder `[@declarator@]` as follows:

```

main ()
{
    struct
    {
->        char    {@declarator@} [>{@constant_expression@}];

            [member_decl@] ...
            }    [init_declarator@] ... ;
    [block_decl@] ...

    {@statement@} ...
    }

```

Type the value `city` over `{@declarator@}` and the value `20` over `[@constant_expression@]` as follows:

```

main ()
{
    struct
    {
->        char    city [20];
            [member_decl@] ...
            }    [init_declarator@] ... ;
    [block_decl@] ...

    {@statement@} ...
    }

```

Expand the placeholder `[@member_decl@]` again to `[@data_type@] {@declarator@} ... ;`. Type the value `int` over `[@data_type@]` and the value `population` over `{@declarator@}`. Erase the placeholder `{@declarator@}` ... as follows:

```

main ()
{
    struct
    {
->        char    city [20];
            int    population;
            [member_decl@] ...
            }    [init_declarator@] ... ;
    [block_decl@] ...

    {@statement@} ...
    }

```

Erase the list placeholder `[@member_decl@]` and expand `[@init_declarator@]` as follows:

```

main ()
{
    struct
    {
->        char    city [20];
            int    population;
            }    {@declarator@} [>= initializer@], [init_declarator@] ... ;
    [block_decl@] ...

    {@statement@} ...
    }

```


Expand `{@declarator@}` to display a menu and select the option `{@declarator@}` `[[@constant_expression@]]` as follows:

```
main ()
{
    struct
    {
        char    city [20];
        int     population;
    }    {@declarator@} [[@constant_expression@]] [= initializer@],
    {@init_declarator@} ... ;
    {@block_decl@} ...
    {@statement@} ...
}
```

Type the value data over `{@declarator@}` and the value 2 over `[@constant_expression@]` as follows:

```
main ()
{
    struct
    {
        char    city [20];
        int     population;
    }    data [2] [= initializer@],
    {@init_declarator@} ... ;
    {@block_decl@} ...
    {@statement@} ...
}
```

Expand the placeholder `[@= initializer@]` to display a menu and select the option = `{@init_multiple_line_form@}` as follows:

```
main ()
{
    struct
    {
        char    city [20];
        int     population;
    }    data [2] = {@init_multiple_line_form@},
    {@init_declarator@} ... ;
    {@block_decl@} ...
    {@statement@} ...
}
```

Expand `{@init_multiple_line_form@}` to produce the following:

```
main ()
{
    struct
    {
        char    city [20];
        int     population;
        } data [2] = {
->                {@init_item@} ...
                }, {@init_declarator@} ... ;
    {@block_decl@} ...
    {@statement@} ...
}
```

Expand `{@init_item@}` to display a menu and select the option `{ {@init_item@} ... }` as follows:

```
main ()
{
    struct
    {
        char    city [20];
        int     population;
        } data [2] = {
->                { {@init_item@} ... }, {@init_item@} ...
                }, {@init_declarator@} ... ;
    {@block_decl@} ...
    {@statement@} ...
}
```

Type the value “Boston” over `{@init_item@}` as follows:

```
main ()
{
    struct
    {
        char    city [20];
        int     population;
        } data [2] = {
->                { "Boston", {@init_item@} ... },
                {@init_item@} ...
                },
                {@init_declarator@} ... ;
    {@block_decl@} ...
    {@statement@} ...
}
```

Expand the optional placeholder `[@init_item@]` to display a menu and select the option `{@init_constant_expression@}` as follows:

```

main ()
{
    struct
    {
        char    city [20];
        int     population;
    }    data [2] = {
->        { "Boston", {@init_constant_expression@}, {@init_item@} }
        {@init_item@} ...
    },
    {@init_declarator@} ... ;
[@block_decl@] ...
{@statement@} ...
}

```

Type the value 250000 over **{@init_constant_expression@}** and erase the immediately following **[@init_item@]** placeholder as follows:

```

main ()
{
    struct
    {
        char    city [20];
        int     population;
    }    data [2] = {
->        { "Boston", 250000 },
        {@init_item@} ...
    },
    {@init_declarator@} ... ;
[@block_decl@] ...
{@statement@} ...
}

```

Expand the list placeholder **[@init_item@]** in the same manner to produce the following:

```

main ()
{
    struct
    {
        char    city [20];
        int     population;
    }    data [2] = {
->        { "Boston", 250000 },
        { "Manchester", 25000 }
    },
    {@init_declarator@} ... ;
[@block_decl@] ...
{@statement@} ...
}

```

C.1.3.5 Statements and Expressions

```
->      [@function_definition@] . . .
```

Expand the placeholder `[@function_definition@]` to display a menu and select the option `{main_function_def}` as follows:

```
->      [@function_level_comments@]
      {@main ( ) OR main function that accepts arguments from the command line@}
      {
        [@block_decl@] . . .
        {@statement@} . . .
      }
      [@function_definition@] . . .
```

Erase the placeholder `[@function_level_comments@]` and the duplicated list placeholder `[@function_definition@]`. Expand `{@main () OR main function that accepts arguments from the command line@}` to display a menu and select the option `main ()` as follows:

```
->      main ( )
      {
        [@block_decl@] . . .
        {@statement@} . . .
      }
```

Expand the placeholder `{@statement@}` to display a menu and select the option `if` as follows:

```
      main ( )
      {
        [@block_decl@] . . .
->      if ({@expression@})
        {
          {@statement@}
          [@else if (expression) statement@]
          [@else statement@]
          [@statement@] . . .
        }
      }
```

Erase the optional placeholders `[@else if (expression) statement@]` and `[@else statement@]`. Expand the placeholder `{@expression@}` to display a menu and select the option `{@binary_expression@}` as follows:

```
      main ( )
      {
        [@block_decl@] . . .
->      if ({@binary_expression@})
        {
          {@statement@}
          [@statement@] . . .
        }
      }
```

Another menu is automatically displayed; select the option {@expression@} {@ {<,>, <=,>=, ==, !=} @} {@expression@} as follows:

```
main ()
{
    [:@block_decl@] ...
->    if ({@expression@} {@ {<,>, <=,>=, ==, !=} @} {@expression@})
        {@statement@}
    [:@statement@] ...
}
```

Type the value count over {@expression@}. Expand the placeholder {@ {<,>, <=,>=, ==, !=} @} to display a menu and select the option < as follows:

```
main ()
{
    [:@block_decl@] ...
->    if (count < {@expression@})
        {@statement@}
    [:@statement@] ...
}
```

Type the value 10 over {@expression@} as follows:

```
main ()
{
    [:@block_decl@] ...
->    if (count < 10)
        {@statement@}
    [:@statement@] ...
}
```

Expand {@statement@} to display a menu and select the option {@expression@}; as follows:

```
main ()
{
    [:@block_decl@] ...
->    if (count < 10)
        {@expression@};
    [:@statement@] ...
}
```

Expand {@expression@} to display a menu and select the option primary. Another menu is automatically displayed. Select the option function_call to produce the following:

```

main ()
{
    [:@block_decl@] ...
    if (count < 10)
->     {:@primary@} ([:@actual_argument@] ... );
    [:@statement@] ...
}

```

Type the value `printf` over `{:@primary@}`, and expand `{:@actual_argument@}` as follows:

```

main ()
{
    [:@block_decl@] ...
    if (count < 10)
->     printf ({:@expression@}, [:@actual_argument@] ... );
    [:@statement@] ...
}

```

Expand `{:@expression@}` to display a menu and select the option `primary` again as follows:

```

main ()
{
    [:@block_decl@] ...
    if (count < 10)
->     printf ({:@primary@}, [:@actual_argument@] ... );
    [:@statement@] ...
}

```

Another menu is automatically displayed; select the option `{:@string_text@}` as follows:

```

main ()
{
    [:@block_decl@] ...
    if (count < 10)
->     printf ("{:@string_text@}", [:@actual_argument@] ... );
    [:@statement@] ...
}

```

Type the string “less than %d test cases\n” over `{:@string_text@}` as follows:

```

main ()
{
    [:@block_decl@] ...
    if (count < 10)
->     printf ("less than %d test cases\n", [:@actual_argument@] ... );
    [:@statement@] ...
}

```

Type the value count over `[@actual_argument@] . . .` and erase the duplicated placeholder `[@actual_argument@] . . .` as follows:

```
main ()
{
    [@block_decl@] . . .
    if (count < 10)
->     printf ("less than %d test cases\n", count);
    [@statement@] . . .
}
```

C.2 Using the VAX Source Code Analyzer

The VAX Source Code Analyzer (SCA) is an interactive source code cross-reference and static analysis tool that works with most VAX programming languages. SCA helps developers keep track of the details of complex, large-scale software systems by displaying source information in response to your queries. SCA uses data generated by the VAX C compiler to supply the requested source information. That information is stored in the SCA library. The data in an SCA library consists of the names of, and information about, all the symbols, modules, and files encountered during a specific compilation of the source.

SCA has both cross-reference and static analysis query features. Cross-referencing supplies information about program symbols and source files. Cross-referencing includes the following features:

- Locating names and occurrences (uses) of these names
- Querying a specified set of names or partial names (you can use wildcards)
- Limiting a query to specific characteristics (such as routine names, variable names, or source files)
- Limiting a query to specific occurrences (such as the primary declaration of a symbol, read or write occurrences of a symbol, or occurrences of a file)

The static analysis query features of SCA provide structural information on the interrelation of routines, symbols, and files. Static analysis includes the following features:

- Displaying routine calls to and from a specified routine
- Analyzing routine calls for consistency as to the numbers and data types of arguments passed, and the types of values returned

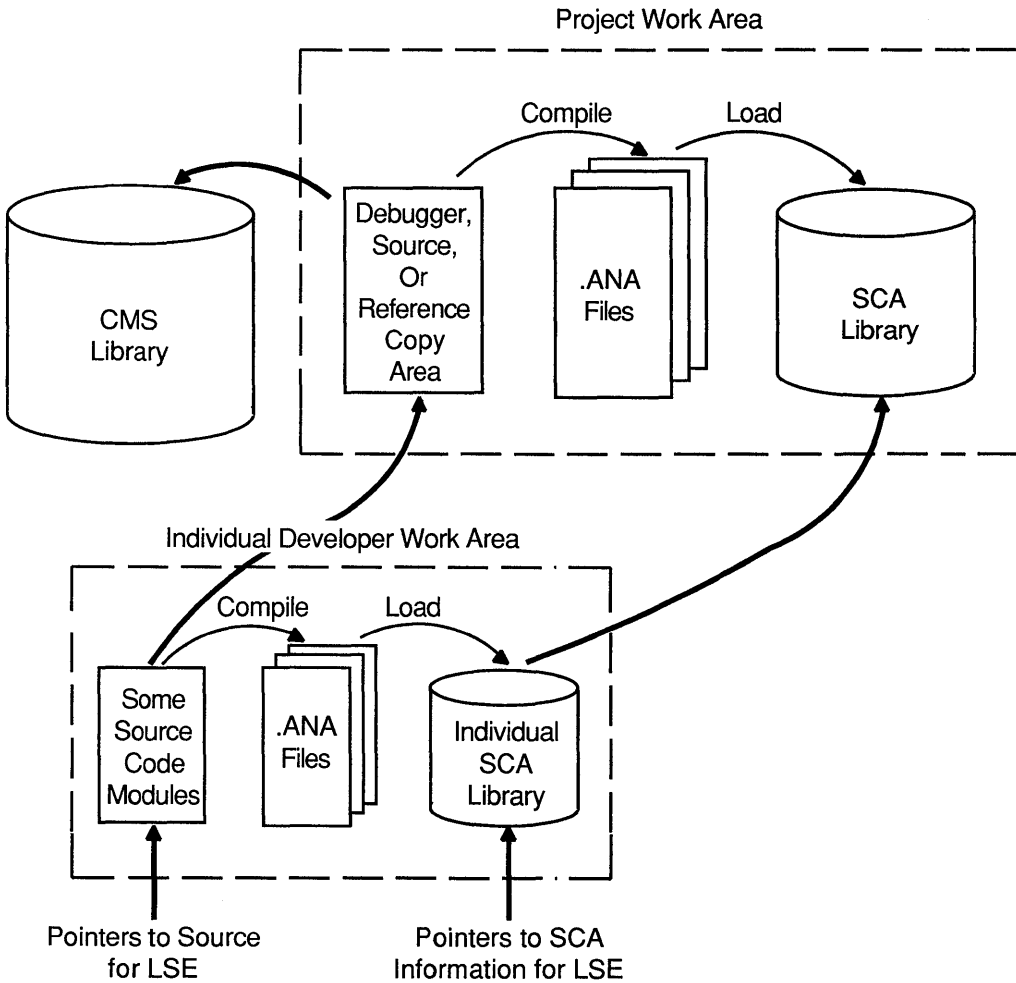
SCA is fully integrated with LSE to provide extended features. By using SCA with LSE, you can view any portion of an entire system and edit related source files.

The following sections provide a general overview of SCA and discuss some of the commands that are available to you while using SCA within LSE. For detailed information on SCA and its use with various programming languages, see the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer*.

C.2.1 Multimodular Development

The cross-referencing and static analysis features of SCA can be useful during the implementation and maintenance phases of a project that involves many programming modules. For example, Figure C-1 shows a project team work area that contains a set of source modules. To keep track of these modules in their various development stages, the team can use a code management tool, such as VAX DEC/Code Management System (CMS), which is represented in the figure by the CMS Library.

Figure C-1: Use of SCA for Multimodular Development



ZK-5850-GE

When the team compiles the source code, an `/ANALYSIS_DATA` qualifier to the `COMPILE` command instructs the `VAX C` compiler to generate SCA-required source information (`.ANA` data files) from the sources. The team then instructs SCA to load the `.ANA` files into a previously established SCA library.

When a team member wants to do additional development work on specific modules, that member sets up an individual work area. Such individual work areas might consist of the following:

- Copies of source and object modules from the project libraries
- Local SCA libraries that contain copies of the module information required to complete assigned tasks

To make the module-viewing capabilities of SCA and LSE integration available, the project team member must inform LSE of the locations of the latest sources, and the related source information. The team member provides pointers to these locations by supplying a search list for LSE.

The search list first points to source modules in individual team member's default directories, and then points to the remaining modules in the project source directory. With such an arrangement, each member can effectively "see" through the local work area to the project-wide area. If an individual work area contains only new modules, and all the work can be done with local resources, the team member need not specify the pointers to the project-wide area.

C.2.2 Setting Up an SCA Environment

To set up an SCA environment, you must take the following steps:

1. Create an SCA library in a subdirectory.
2. Select the library.
3. Use the VAX C compiler to generate the data analysis (.ANA) files for each source module in your system.
4. Load these data analysis files into your local SCA library.

You can now use SCA to conduct source information queries.

C.2.2.1 Creating an SCA Library

To use SCA, you must have an SCA library to store the detailed source analysis data that the VAX C compiler collects. Source analysis data is information about all of the symbols, files, and modules contained in the source.

To create an SCA library, you first create a subdirectory at the DCL level. For example:

```
$ CREATE/DIRECTORY PROJ:[USER.LIB1]
```

This command creates a subdirectory LIB1 for a local SCA library.

To initialize a new SCA library, specify the CREATE LIBRARY command. This command has the following form:

```
CREATE LIBRARY [/qualifiers ... ] directory-spec[, ... ]
```

For example:

```
$ SCA CREATE LIBRARY [.LIB1]
```

This command initializes and activates library LIB1.

C.2.2.2 Generating the Data Analysis Files

SCA uses detailed source data that is generated by the VAX C compiler. When you specify the /ANALYSIS_DATA qualifier on the CC command line, the generated data is output to a file with the default type .ANA. For example:

```
$ CC/LIST/DIAGNOSTICS/ANALYSIS_DATA PG1,PG2,PG3
```

This command line compiles the input files PG1.C, PG2.C and PG3.C, and generates four corresponding output files for each input file. The compiler puts these files in your current default directory unless you specify otherwise.

C.2.2.3 Selecting an SCA Library

To select an existing SCA library to use with your current SCA session, use the SET LIBRARY command. This command has the following form:

```
SET LIBRARY [/qualifiers ... ] directory-spec[, ... ]
```

A message appears in the message buffer, at the bottom of your screen, indicating whether your SCA library selection was successful or not.

C.2.2.4 Loading Data Analysis Files into a Local Library

Before you can examine the information in the compiler-generated source analysis (.ANA) files, you must load the files into an SCA library using the LOAD command. This command has the following form:

```
LOAD [/qualifiers ... ] file-spec[, ... ]
```

For example:

```
LSE> LOAD PG1,PG2,PG3
```

This command loads your library with the modules contained in the data analysis files PG1, PG2, and PG3.

C.2.3 Using SCA for Cross-Referencing

With an SCA library in place, you can ask for symbol or file information by using the SCA command `FIND`. This command has the following form:

```
FIND [/qualifier . . . ] [name-expression[ . . . ]]
```

The name-expression is the name of a symbol or file. It can be explicit (such as `ABC`), it can include wildcards (such as `ABC*` or `AB%`), and it can include more than one name by specifying a list of name expressions separated by commas. For example:

```
LSE> FIND ABC,XY%
```

VAX C reports symbols to SCA in all uppercase. Therefore, the variables `abc` and `ABC` will both be found by the previous command.

You can query SCA library information for the following data that exist within a source program:

Name	A series of characters that uniquely identify a symbol or a file.
Item	An appearance of a symbol (such as a variable, constant, label, or procedure) or a file.
Occurrence	The use of a symbol or a file.

To limit the information resulting from a query, use qualifiers on the `FIND` command, such as the `/DECLARATIONS` and `/REFERENCE` qualifiers. For example:

```
LSE> FIND/REFERENCES=CALL BUILD_TABLE
```

This command causes SCA to report only references in the source code where the routine `BUILD_TABLE` is called.

When you first enter a `FIND` command within LSE, you initiate a query session. Within this context, the integration of LSE and SCA provides the commands listed in Table C-4, which can only be used within LSE.

Table C-4: SCA Commands to Use Within LSE

Command	Description
{ NEXT PREVIOUS }	{ NAME ITEM OCCURRENCE QUERY STEP }
GOTO SOURCE	Closely associated commands that let you step through one or more query buffer displays within LSE.
GOTO DECLARATION	Displays the source corresponding to the current query item.
	Positions the cursor on a symbol declaration in one window, and displays the source code that contains the symbol declaration in another window.

Language Summary

This appendix briefly describes the CC and LINK commands of the DIGITAL Command Language (DCL) and the qualifiers used with both commands. This appendix also briefly describes C language features.

This appendix presents the following syntax summaries:

- The CC command (Section D.1)
- The LINK command (Section D.2)
- Data-type keywords (Section D.3)
- Precedence of operators (Section D.4)
- Statements (Section D.5)
- Conversion rules (Section D.6)
- Escape sequences (Section D.7)
- Preprocessor directives (Section D.8)
- Record Management Services—RMS (Section D.9)

D.1 The CC Command

The DCL command CC compiles one or more VAX C source files into one or more object files. The source file or files compiled into an object module is called the *compilation unit*.

Syntax:

CC[qualifier . . .] file-spec-list

File Specification Syntax:

file-spec[qualifier . . .]
file-spec-list, file-spec[qualifier . . .]
file-spec-list + file-spec[qualifier . . .]

Command Qualifiers

/[NO]ANALYSIS_DATA[=file-spec]
/[NO]CROSS_REFERENCE
/[NO]DEBUG[=option]
/[NO]DEFINE[=(definition list)]
/[NO]DIAGNOSTICS[=file-spec]
/G_FLOAT
/[NO]INCLUDE_DIRECTORY=(pathname [, . . .])
/LIBRARY
/[NO]LIST[=file-spec]

/[NO]MACHINE_CODE[=option]
/[NO]OBJECT[=file-spec]
/[NO]OPTIMIZE[=option]
/[NO]PARALLEL
/[NO]PRECISION={SINGLE,DOUBLE}
/[NO]PREPROCESS_ONLY[=filename]
/SHOW[=(option, . . .)]

/[NO]STANDARD=OPTION
/[NO]UNDEFINE[=(undefine list)]
/[NO]WARNINGS[=(option-list)]

Defaults

/NOANALYSIS_DATA
/NOCROSS_REFERENCE
/DEBUG=TRACEBACK
/NODEFINE
/NODIAGNOSTICS
/NOG_FLOAT
/NOINCLUDE_DIRECTORY
None
/NOLIST (interactive mode)
/LIST (batch mode)
/NOMACHINE_CODE
/OBJECT
/OPTIMIZE
/NOPARALLEL
/PRECISION=DOUBLE
/NOPREPROCESS
/SHOW=(NOBRIEF,
NODECOMPOSITION,
NODICTIONARY,
NOEXPANSION,
NOINCLUDE,
NOINTERMEDIATE,
NOSTATISTICS,
NOSYMBOLS,
NOTRANSLATION,
SOURCE,
TERMINAL)
/STANDARD
/NOUNDEFINE
/WARNINGS

NOTE

The only qualifier that *must* be used with a file specification is the /LIBRARY qualifier. You cannot place this qualifier on the CC command line.

D.2 The LINK Command

The DCL LINK command combines one or more object modules into one image file.

Syntax:

```
LINK[/qualifier . . . ] file-spec[/qualifier . . . ], . . .
```

Command Qualifiers	Defaults
/BRIEF	None
/[NO]CONTIGUOUS	/NOCONTIGUOUS
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG[=file_spec]	/NODEBUG
/[NO]EXECUTABLE[=file_spec]	/EXECUTABLE
/FULL	None
/HEADER	None
/[NO]MAP[=file_spec]	/NOMAP
/POIMAGE	None
/PROTECT	None
/[NO]SHAREABLE[=file_spec]	/NOSHAREABLE
/[NO]SYMBOL_TABLE[=file_spec]	/NOSYMBOL_TABLE
/[NO]SYSLIB	/SYSLIB
/[NO]SYSSHR	/SYSSHR
/[NO]SYSTEM[=base_address]	/NOSYSTEM
/[NO]TRACEBACK	/TRACEBACK
/[NO]USERLIBRARY[=table[, . . .]]	None
/INCLUDE=(module_name[, . . .])	None
/LIBRARY	None
/OPTIONS	None
/SELECTIVE_SEARCH	None

NOTE

The only qualifiers that *must* be used with a file specification are the /INCLUDE, /LIBRARY, /OPTIONS, and /SELECTIVE_SEARCH qualifiers. You cannot place these qualifiers on the LINK command.

D.3 Data-Type Keywords

Type Specifiers:

32-bit signed or unsigned:

int
long
long int
unsigned int
unsigned long
unsigned long int

16-bit signed or unsigned:

short
short int
unsigned short
unsigned short int

8-bit signed or unsigned:

char
unsigned char

F_floating format:

float

D_floating or G_floating format:

double
long float

Aggregate types:

struct
union
variant_struct
variant_union

Enumerated type:

enum

Type of function return value:

void

Type declaration:

typedef

Storage-class specifiers:

auto
register
static
extern
globaldef
globalref
globalvalue

Data-type modifiers:

const
volatile

Storage-class modifiers:

readonly
noshare
_align

D.4 Precedence of Operators

Table D-1 lists the operators from highest precedence to lowest. In the binary operator category, operators appear in descending order of precedence, line by line.

Table D-1: Precedence of Operators

Category	Association	Operator
Primary	Left to right	() [] -> .
Unary	Right to left	! ~ ++ -- (type) - * & sizeof

(continued on next page)

Table D-1 (Cont.): Precedence of Operators

Category	Association	Operator
Binary	Left to right	* / % + - << >> < <= > >= == != & ^ &&
Conditional	Right to left	?:
Assignment	Right to left	= += -= *= /= %= > >= < <= &= ^= =
Comma	Left to right	,

D.5 Statements

Syntax:

[expression] ;

identifier : *statement*

{ [*declaration-list*] [*statement-list*] }

case *constant-expression* default: *statement-list*

if *expression*) *statement* [**else** *statement*]

while *expression*) *statement*

do *statement* **while** (*expression*)

for ([*expression*] ; [*expression*] ; [*expression*])
statement

switch (*expression*) *statement*

break ;

continue ;

return [*expression*] ;

goto *identifier* ;

D.6 Conversion Rules

Arithmetic Conversion

Any operand of type:	Is converted to:
char	int
short	int
unsigned char	unsigned int
unsigned short	unsigned int
float	double

If operand type is:	The result and the other operands are:
double	double
unsigned	unsigned

Otherwise, both operands are:	And the result is:
int	int

Function Argument Conversion

Any argument of type:	Is converted to type:
float	double ¹
char	int
short	int
unsigned char	unsigned int
unsigned short	unsigned int
array	pointer to array
function	pointer to function

¹If the compiler encounters a **float** in an argument to a function declared with a function prototype, then the argument remains a **float**.

D.7 Escape Sequences

Table D-2: Escape Sequences

Character	Mnemonic	Escape Sequence
newline	NL	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
apostrophe	'	\'
quotes	"	\"
bit pattern	ddd	\ddd or \xddd

Use the form “\ddd” to specify any byte value (usually an ASCII code), where the digits ddd are one to three octal digits. The octal digits are limited to 0 through 7.

Similarly, use the form “\xddd” to specify any byte value (usually an ASCII code), where the digits are used to specify one to three hexadecimal digits.

D.8 Preprocessor Directives

Syntax:

#define *identifier*[(*param1*, . . . *param2*)] *token-string*

#undef *identifier*

#dictionary *cdd-path*

#elif *constant-expression*

#include <*file-spec*>

#include “*file-spec*”

#include *module-name*

#if *constant-expression*

#ifdef *identifier*

```

#ifndef identifier

#else

#endif

#[line] constant string
#[line] constant identifier

#module identifier identifier
#module identifier string

#pragma [(identifier[, . . . ])]

```

D.9 Record Management Services (RMS)

The RMS functions can be expressed in terms of the following general descriptions:

Syntax:

```

#include (rms-module)
int sys$name(pointer, [error_function],
             [success_function])

struct rms_structure *pointer;
int (*error_function)(), (*success_function)();

```

rms-module

Is the name of one of the modules in Table D–3.

Table D–3: RMS Module Names

Module	Description	Structure Tag
<i>fab</i>	File access block	FAB
<i>rab</i>	Record access block	RAB
<i>nam</i>	Name block	NAM
<i>xaball</i>	Allocation XAB	XABALL
<i>xabdat</i>	Date and time XAB	XABDAT
<i>xabfhc</i>	File header characteristics XAB	XABFHC

(continued on next page)

Table D–3 (Cont.): RMS Module Names

Module	Description	Structure Tag
<i>xabkey</i>	Indexed file key XAB	XABKEY
<i>xabpro</i>	Protection XAB	XABPRO
<i>xabrdt</i>	Revision date and time XAB	XABRDT
<i>xabsum</i>	Summary XAB	XABSUM
<i>xabtrm</i>	Terminal control XAB	XABTRM
<i>rmsdef</i>	Completion status codes	—
<i>rms</i>	All RMS modules	All tags

sys\$name

Is the name of the RMS function being called.

pointer

Is a pointer to an RMS structure that (optionally) has been initialized by the templates in Table D–4.

Table D–4: RMS Templates

Template	Description
<i>cc\$rms_fab</i>	Initializes the file access block (FAB)
<i>cc\$rms_rab</i>	Initializes the record access block (RAB)
<i>cc\$rms_nam</i>	Initializes the name block (NAM)
<i>cc\$rms_xaball</i>	Initializes the allocation XAB (XABALL)
<i>cc\$rms_xabdat</i>	Initializes the date and time XAB (XABDAT)
<i>cc\$rms_xabfhc</i>	Initializes the file header characteristics XAB (XABFHC)
<i>cc\$rms_xabkey</i>	Initializes the indexed file key XAB (XABKEY)
<i>cc\$rms_xabpro</i>	Initializes the protection XAB (XABPRO)
<i>cc\$rms_xabrdt</i>	Initializes the revision date and time XAB (XABRDT)
<i>cc\$rms_xabsum</i>	Initializes the summary XAB (XABSUM)
<i>cc\$rms_xabtrm</i>	Initializes the terminal control XAB (XABTRM)

error_function

Is the name of a signal-handling function to call if an error occurs (optional).

success_function

Is the name of a function to call if the RMS function is successful (optional).

rms_structure

Is the type of structure being pointed to by pointer.

The RMS functions return an integer status value.

Working with the Multiprocess Debugging Configuration

This appendix assumes that you are familiar with the configuration of the VMS Debugger that supports single-process debugging (referred to in this appendix as the default debugging configuration). It covers only the extensions that are provided to support multiprocess debugging. (See Chapter 2 for information about how to use the debugger for single-process debugging.)

This appendix provides information in the following areas:

- Basic information that you need to perform multiprocess debugging (Section E.1)
- Supplemental information on more advanced concepts and usage than those described in Section E.1 (Section E.2)
- A sample multiprocess debugging session (Section E.3)
- System management considerations associated with multiprocess debugging (Section E.4)

The information in this appendix is oriented toward multiprocess debugging in general, not toward VAX C parallel-processing debugging. For more detailed information about the parallel-processing debugging commands, see the debugger HELP utility or the *VMS Debugger Manual*.

E.1 Getting Started

This section gives a quick overview of the multiprocess debugging environment, by running through the basic steps and commands. Later sections are referenced for additional details. See the debugger's online HELP for complete details on commands.

E.1.1 Establishing a Multiprocess Debugging Configuration

Before invoking the debugger, enter the following command to establish a multiprocess configuration:

```
$ DEFINE/JOB DBG$PROCESS MULTIPROCESS
```

This command establishes a multiprocess configuration for the job tree in which the command was issued. As a result, once a debugging session is started, any image that can be debugged running in the same job tree can be controlled from that one session. (An image can be debugged if it has been compiled and linked with the /DEBUG qualifier.)

E.1.2 Invoking the Debugger

This section explains how to start a multiprocess debugging session. See Section E.2.3 for additional techniques for invoking the debugger (for example, using the CONNECT command or a CTRL/Y-DEBUG sequence).

You typically initiate the execution of a multiprocess program by running the main image in the main (master) process. Once the main image is running in the main process, the program will spawn one or more subprocesses to run additional images by issuing a LIB\$SPAWN run-time library call or a \$CREPRC system service call. (VAX C performs this step during the initialization phase.)

If the main image can be debugged, the debugger is invoked when you run the image. For example:

```
$ RUN MAIN_PROG
      VAX DEBUG Version X5.0-3 MP

%DEBUG-I-INITIAL, language is C, module set to MAIN_PROG
%DEBUG-I-NOTATMAIN, type GO to get to start of main program
predefined trace on activation at routine MAIN_PROG in %PROCESS_NUMBER 1
DBG_1>
```

As with a one-process program, the debugger displays its banner and prompt just prior to executing the main image. However, note two differences—the “predefined trace on . . .” message and the debugger prompt.

In a multiprocess configuration, the debugger traces each new process that is brought under control. In this case, the debugger traces the first process, which runs the main image of the program. (%PROCESS_NUMBER is a built-in symbol that identifies a process number, just as %LINE identifies a line number.)

The significance of the prompt suffix (_1) is explained in the next section.

E.1.3 The Visible Process and Process-Specific Commands

The previous example shows that the debugger prompt in a multiprocess debugging configuration is different from that found in the default configuration.

In a multiprocess configuration, “dynamic prompt setting” is enabled (SET PROMPT/SUFFIX=PROCESS_NUMBER) by default. Therefore, the prompt has a process-specific suffix that indicates the process number of the *visible* process. The debugger assigns a process number sequentially, starting with process 1, to each process that comes under the control of a given debugging session.

The visible process is the process that is the default context for issuing process-specific commands. *Process-specific commands* are those that start execution (STEP, GO, and so on) and those used for looking up symbols, setting breakpoints, looking at the call stack and registers, and so on. Commands that are not process specific are those that do not depend on the mapping of virtual memory but, rather, affect the entire debugging environment (for example, keypad mode and screen mode commands).

Unless dynamic prompt setting is disabled (using the SET PROMPT/NOSUFFIX command), the debugger prompt suffix identifies the visible process (for example, DBG_1>). The SET PROMPT command provides several options for tailoring the prefix and suffix of the prompt string to your needs.

E.1.4 Obtaining Information About Processes

Use the SHOW PROCESS command to obtain information about processes that are currently under control of your debugging session. By default, SHOW PROCESS displays one line of information about the visible process. The following example shows the kind of information displayed immediately after you invoke the debugger:

```
DBG_1> SHOW PROCESS
  Number Name           Hold State           Current PC
*    1 JONES             activated        MAIN_PROG\%LINE 2
DBG_1>
```

A one-line SHOW PROCESS display provides the following information about each process specified:

- The process number assigned by the debugger. In this case, the process number is 1 because this is the first process known to the debugger. The asterisk in the leftmost column (*) marks the visible process.
- The VMS process name. In this case, the VMS process name is JONES.

- Whether the process has been placed on hold with a SET PROCESS/HOLD command (see Section E.1.7.2). In this case, the process has not been placed on hold.
- The current debugging state for that process. A process is in the activated state when it is first brought under debugger control (that is, before it has executed any part of the program under debugger control). Table E-1 summarizes the possible debugging states that may appear in the state column.
- The location (symbolized, if possible) where execution of the image is suspended in that process. In this case, the image has not started execution.

Table E-1: Debugging States

Activated	The image and its process have just been brought under debugger control, either through a DCL RUN/DEBUG command, a debugger CONNECT command, a CTRL/Y-DEBUG sequence, or by the program signaling SS\$DEBUG while it was not under debugger control.
Break ¹	A breakpoint was triggered.
Interrupted	Execution was interrupted in that process, either because execution was suspended in some other process or because the user interrupted program execution with the abort-key sequence (CTRL/C, by default).
Step ¹	A STEP command has completed.
Terminated	The image has terminated execution but the process is still under debugger control. Therefore, you can obtain information about the image and its process.
Trace ¹	A tracepoint was triggered.
Unhandled exception	An unhandled exception was encountered.
Watch of	A watchpoint was triggered.

¹See the SHOW PROCESS command in the DCL command dictionary for a list of additional states.

The SHOW PROCESS/ALL command provides information about all processes that are currently under debugger control. In the case of the previous example, a SHOW PROCESS/ALL command would show only process 1. The SHOW PROCESS/FULL command provides additional details about processes.

E.1.5 Bringing a Spawned Process Under Debugger Control

This section describes, in general, how the debugger interacts with spawned processes.

NOTE

Most of the information in this section is not pertinent to the debugging of parallel loops. The connect operations described in this section are automatically performed for you during the initialization phase.

To show the interaction, assume that you are entering a few STEP commands and, in the middle of a step, MAIN_PROG spawns a process to run an image that can be debugged called TEST.

Because DBG\$PROCESS has the value MULTIPROCESS, the spawned process is now requesting to connect to the current debugging session, and image TEST is suspended at the start of execution.

While the spawned process is waiting to be connected, it is not yet known to the debugger and cannot be identified in a SHOW PROCESS/ALL display. You can bring the process under debugger control using either of the following methods:

- Enter a command, such as STEP, that starts execution.
- Enter the CONNECT command without specifying a parameter. The CONNECT command is preferable in those cases when you do not want the program to execute any further.

The following example shows how to use the CONNECT command:

```
DBG_1> STEP
stepped to MAIN_PROG\%LINE 18 in %PROCESS_NUMBER 1
18:      LIB$SPAWN("RUN/DEBUG TEST");
DBG_1> STEP
stepped to MAIN_PROG\%LINE 21 in %PROCESS_NUMBER 1
21:      X = 7;
DBG_1> CONNECT
predefined trace on activation at routine TEST in %PROCESS_NUMBER 2
DBG_1>
```

In this example, the second STEP command takes you past the LIB\$SPAWN call that spawns the process. The CONNECT command brings the waiting process under debugger control. After entering the CONNECT command, you may need to wait a moment for the process to connect. The “predefined trace on . . .” message, as explained in Section E.1.2, indicates that the debugger has taken control of a new process and identifies that process as process 2, the second process known to the debugger in this session.

A `SHOW PROCESS/ALL` command, entered at this point, identifies the debugging state for each process and the location at which execution is suspended:

```
DEG_1> SHOW PROCESS/ALL
Number Name          Hold State          Current PC
*      1 JONES          step          MAIN_PROG\%LINE 21
      2 JONES_1        activated     TEST\%LINE 1+2
DEG_1>
```

Note that the `CONNECT` command brings any processes that are waiting to be connected to the debugger under debugger control. If no processes are waiting, you can press `CTRL/C` to abort the `CONNECT` command and display the debugger prompt.

E.1.6 Broadcasting Commands to Selected Processes

By default, process-specific commands are executed in the context of the visible process. The `DO` command enables you to execute commands in the context of one or more processes that are currently under debugger control. This capability is referred to as broadcasting commands to processes.

Use the `DO` command without a qualifier to execute commands in the context of all of the processes. For example, the following command executes the `SHOW CALLS` command for all processes that are currently under debugger control (processes 1 and 2, in this case):

```
DEG_1> DO (SHOW CALLS)
For %PROCESS_NUMBER 1
  module name      routine name      line      rel PC      abs PC
  *MAIN_PROG      MAIN_PROG          21        0000001E    0000041E
For %PROCESS_NUMBER 2
  module name      routine name      line      rel PC      abs PC
  TEST             TEST              1+2       0000000B    0000040B
```

Use the `DO` command with the `/PROCESS` qualifier to execute commands in the context of selected processes. For example, the following command executes the commands `SET MODULE START` and `EXAMINE X` in the context of process 2 (see Section E.2.1 for information on how to specify processes in debugger commands):

```
DEG_1> DO/PROCESS=(%PROC 2) (SET MODULE START; EXAMINE X)
```

E.1.7 Controlling Execution

Program execution in a multiprocess debugging environment follows these conventions:

- When you enter a command that starts program execution, such as STEP or GO, the command is executed in the context of the visible process. However, images in any other unheld processes (processes that have not been placed on hold with a SET PROCESS/HOLD command) are also allowed to execute. Similarly, if you use the DO command to broadcast a command to start execution in one or more processes, the command is executed in the context of each specified unheld process, but images in any other unheld processes are also allowed to execute. In all cases, a hold condition is ignored in the visible process. (See Section E.1.7.2 for additional information about the behavior of processes when on hold.)
- Once execution is started, the way in which it continues depends on whether the command SET MODE [NO]INTERRUPT was entered. By default (SET MODE INTERRUPT), execution continues until it is suspended in any process. At that point, execution is interrupted in any other processes that were executing images, and the debugger prompts for input.

These concepts are shown by continuing with the example in Section E.1.5, which shows the two STEP commands.

In that example, the “stepped to . . . ” messages indicate that both commands are executed in the context of process 1, the visible process. The second STEP command spawns process 2. The SHOW PROCESS/ALL example of Section E.1.5 indicates that execution in processes 1 and 2 is suspended at MAIN_PROG\%LINE 21 and TEST\%LINE 1+2, respectively.

At this point, entering another STEP command followed by SHOW PROCESS/ALL results in the following display:

```
DBG_1> STEP
stepped to MAIN_PROG\%LINE 23 in %PROCESS_NUMBER 1
23:          Y = 15;
DBG_1> SHOW PROCESS/ALL
Number Name           Hold State           Current PC
*   1 JONES            step             MAIN_PROG\%LINE 23
   2 JONES_1          interrupted      TEST\%LINE 3+1
DBG_1>
```


The STEP command is executed in the context of process 1, the visible process. After the STEP, execution in process 1 is suspended at MAIN_PROG\%LINE 23. However, the STEP command also causes execution to start in process 2. The completion of the STEP in process 1 causes execution in process 2 to be interrupted at TEST\%LINE 3+1.

Section E.1.7.1 describes another mode of execution, which is provided by the SET MODE NOINTERRUPT command.

E.1.7.1 Controlling Execution with SET MODE NOINTERRUPT

The SET MODE NOINTERRUPT command allows execution to continue without interruption in other processes when it is suspended in some process. This is especially useful if, for example, you want to broadcast a STEP command to several processes with the DO command and complete execution of the STEP in all these processes. For example:

```
DBG_1> SET MODE NOINTERRUPT
DBG_1> DO (STEP)
```

In this example, the DO command executes the STEP command in the context of all processes. The visible process and any other unheld processes start execution. Because the command SET MODE NOINTERRUPT was entered, the prompt is displayed only after the STEP has completed (or execution has been otherwise suspended at a breakpoint or watchpoint) in all processes that were executing.

When SET MODE NOINTERRUPT is in effect, as long as execution continues in any process, the debugger does not prompt for input. In such cases, use CTRL/C to interrupt all processes and display the prompt.

E.1.7.2 Putting Selected Processes on Hold

As indicated in the previous sections, a command that starts execution is executed in the context of the visible process, but it also causes execution to start in other processes. If you want to inhibit execution in a process, put it on hold. For example, the following SET PROCESS/HOLD command puts process 2 on hold. The subsequent STEP command is executed in the context of process 1, the visible process. Execution also starts in any other processes that are not on hold, but not in process 2.

```
DBG_1> SET PROCESS/HOLD %PROC 2
DBG_1> STEP
```

A SHOW PROCESS display indicates whether a process is on hold. For example:

```
DBG_1> SHOW PROCESS/ALL
Number Name           Hold State           Current PC
*   1 JONES              step             MAIN_PROG\%LINE 24
   2 JONES_1            HOLD interrupted   TEST\%LINE 3+1
DBG_1>
```

To unhold a process, enter the SET PROCESS/NOHOLD command, specifying the process that you want to release from the hold condition.

A hold condition is ignored in the visible process. Therefore, the command SET PROCESS/HOLD/ALL is a convenient way to confine execution to the visible process. In the following example, execution starts only in the visible process:

```
DBG_1> SET PROCESS/HOLD/ALL
DBG_1> STEP
```

This feature is useful if, for example, you want to use the CALL command to execute a dump routine that is not part of the execution stream of your program.

The preceding discussions also apply if you use the DO command to broadcast a GO, STEP, or CALL command to several processes. The GO, STEP or CALL command is executed in the context of each specified unheld process, and execution also starts in any other unheld process. The following example shows the execution behavior when all processes are placed on hold and commands are broadcast to all processes. Execution starts only in the visible process (process 1, in this example).

```
DBG_1> SET PROCESS/HOLD/ALL
DBG_1> DO (EXAMINE X: STEP)
For %PROCESS_NUMBER 1
  MAIN_PROG\X: 78
For %PROCESS_NUMBER 2
  TEST\X: 29
stepped to MAIN_PROG\%LINE 26 in %PROCESS_NUMBER 1
26: K = K + 1;
DBG_1>
```

E.1.8 Changing the Visible Process

Use the SET PROCESS command (with the default /VISIBLE qualifier) to establish another process as the visible process. For example, the following command makes process 2 the visible process:

```
DBG_1> SET PROCESS %PROC 2
DBG_2>
```

In this example, because dynamic prompt setting is enabled by default, the SET PROCESS command has also caused the prompt string suffix to change. It now indicates that process 2 is the visible process. All process-specific commands are now executed in the context of process 2. For example, a SHOW CALLS command would display the call stack for the image running in process 2.

E.1.9 Dynamic Process Setting

By default, dynamic process setting is enabled (using the SET PROCESS/DYNAMIC command). As a result, whenever the debugger suspends program execution, the process in which execution is suspended automatically becomes the visible process. Dynamic process setting occurs in the following situations: when a breakpoint or watchpoint is triggered, at an exception condition, on the completion of a STEP command, or when the last process performs an image exit.

When dynamic process setting is disabled (using the /NODYNAMIC qualifier), the visible process remains unchanged until you specify another process with the SET PROCESS/VISIBLE command.

Dynamic process setting is shown in the following example, which also shows dynamic prompt setting:

```
DBG_1> SHOW PROCESS/ALL
Number Name           Hold State           Current PC
*   1 JONES           step           MAIN_PROG\%LINE 22
   2 JONES_1         interrupted    TEST\%LINE 4
DBG_1> DO/PROCESS=(%PROC 2) (SET BREAK %LINE 11)
DBG_1> GO
.
.
.
break at TEST\%LINE 11 in %PROCESS_NUMBER 2
DBG_2> SHOW PROCESS/ALL
Number Name           Hold State           Current PC
*   1 JONES           interrupted    MAIN_PROG\%LINE 28
   2 JONES_1         break         TEST\%LINE 11
DBG_2>
```

In this example, process 1 is initially the visible process, as indicated by the prompt and the SHOW PROCESS display. The DO command sets a breakpoint in the context of process 2. Execution is resumed with the GO command and is suspended at the breakpoint in process 2. Process 2 is now the visible process, as indicated by the prompt and the SHOW PROCESS display.

If you entered the `SET MODE NOINTERRUPT` command and then started execution in several processes with the `DO` command, the prompt would not be displayed until after execution was suspended in all processes. In this case, the visible process remains unchanged, unless the last process performs an image exit (and then becomes the visible process).

E.1.10 Monitoring the Termination of Images

When the main image of a process runs to completion, the process goes into the terminated debugging state. This condition is traced by default, as if you had entered the `SET TRACE/TERMINATING` command.

When a process is in the terminated state, it is still known to the debugger and appears in a `SHOW PROCESS/ALL` display. You can enter commands to examine variables, and so on.

When the last image of the program exits, the debugger gains control and displays its prompt.

E.1.11 Terminating the Debugging Session

To terminate the entire debugging session, use the `EXIT` or `QUIT` command without specifying any parameters. If you do not specify parameters, the behavior of the `EXIT` and `QUIT` commands is similar to their behavior for the default debugging configuration. (`QUIT` does not execute any user-declared exit handlers.)

E.1.12 Releasing Selected Processes from Debugger Control

To release selected processes from debugger control without terminating the debugging session, use the `EXIT` or `QUIT` command, specifying one or more process specifications as parameters. For example, the following command terminates the image running in process 2, and releases the process from debugger control:

```
DBG_3> EXIT %PROC 2
DBG_3>
```

Subsequently, process 2 does not appear in a `SHOW PROCESS` display. See the DCL command dictionary for complete details on the `EXIT` and `QUIT` commands.

E.1.13 Aborting Debugger Commands and Interrupting Program Execution

Use CTRL/C (not CTRL/Y) to abort the execution of a debugger command or to interrupt program execution. This is useful if a command takes a long time to complete or your program loops. Control is returned to the debugger rather than to the DCL command interpreter. For example:

```
DBG_1> GO
.
.
.
[CTRL/C]
%DEBUG-W-ABORTED, command aborted by user request
DBG_1> EXAMINE/BYTE 1000:101000 !should have typed 1000:1010
1000: 0
1004: 0
1008: 0
1012: 0
1016: 0
[CTRL/C]
%DEBUG-W-ABORTED, command aborted by user request
DBG_1>
```

Pressing CTRL/C interrupts execution in every process that is currently running an image. This is indicated as an interrupted state in a SHOW PROCESS display. Pressing CTRL/C also interrupts any debugger command that is currently executing.

If your program has a CTRL/C AST service routine enabled, use the SET ABORT_KEY command to assign the debugger's abort function to another CTRL-key sequence. For example:

```
DBG_1> SET ABORT_KEY = CTRL_P
DBG_1> GO
.
.
.
[CTRL/P]
%DEBUG-W-ABORTED, command aborted by user request
DBG_1> EXAMINE/BYTE 1000:101000 !should have typed 1000:1010
1000: 0
1004: 0
1008: 0
1012: 0
1016: 0
[CTRL/P]
%DEBUG-W-ABORTED, command aborted by user request
DBG_1>
```

Many CTRL-key sequences have VMS predefined functions, and the SET ABORT_KEY command enables you to override such definitions within the debugging session (see the *VMS DCL Concepts Manual*). Some of the CTRL-key characters not used by the VMS operating system are G, K, N, and P.

E.2 Supplemental Information

This section provides details on advanced concepts and usages that relate to multiprocess debugging.

E.2.1 Specifying Processes in Debugger Commands

When specifying processes in debugger commands, you can use any of the forms listed in Table E-2, except when specifying processes with the CONNECT command.

The CONNECT command is used to bring a process that is not yet known to the debugger under debugger control. Therefore, when specifying a process with CONNECT, you can use only its VMS process name or VMS process identification number (PID). You cannot use its debugger-assigned process number or any of the process built-in symbols (for example, %NEXT_PROCESS). (As noted earlier in this appendix, the CONNECT command is not used in the debugging of VAX C parallel loops.)

Table E-2: Process Specifications

<code>[%PROCESS_NAME] process-name</code>	The VMS process name, if that name contains no spaces or lowercase characters. ¹
<code>[%PROCESS_NAME] "process-name"</code>	The VMS process name, if that name contains spaces or lowercase characters. You can also use apostrophes (') instead of quotation marks (").
<code>%PROCESS_PID process_id</code>	The VMS process identification number (PID), which is a hexadecimal number.

¹The process name can include the wildcard character (*).

(continued on next page)

Table E-2 (Cont.): Process Specifications

<code>%PROCESS_NUMBER</code> process-number (or <code>%PROC</code> process-number)	The number assigned to a process when it comes under debugger control. A new number is assigned sequentially, starting with 1, to each process. If a process is released from debugger control (with the <code>EXIT</code> or <code>QUIT</code> command), the number is not reused during the debugging session. Process numbers appear in a <code>SHOW PROCESS</code> display. Processes are ordered in a circular list so they can be indexed with the built-in symbols <code>%PREVIOUS_PROCESS</code> and <code>%NEXT_PROCESS</code> .
process-group-name	A symbol defined with the <code>DEFINE/PROCESS_GROUP</code> command to represent a group of processes.
<code>%NEXT_PROCESS</code>	The next process after the visible process in the debugger's circular process list.
<code>%PREVIOUS_PROCESS</code>	The process previous to the visible process in the debugger's circular process list.
<code>%VISIBLE_PROCESS</code>	The process whose stack, register set, and images are the current context for looking up symbols, register values, routine calls, breakpoints, and so on.

You can omit the `%PROCESS_NAME` built-in symbol when entering commands. For example:

```
DBG_2> SHOW PROCESS %PROC 2, JONES_3
```

You can define a symbol to represent a group of processes (using the `DEFINE/PROCESS_GROUP` command). This enables you to enter commands in abbreviated form. For example:

```
DBG_1> DEFINE/PROCESS_GROUP SERVERS=FILE_SERVER, NETWORK_SERVER
DBG_1> SHOW PROCESS SERVERS
  Number  Name          Hold  State          Current PC
*    1  FILE_SERVER          step          FS_PROG\%LINE 37
    2  NETWORK_SERVER      break         NET_PROG\%LINE 24
DBG_1>
```

The built-in symbols `%VISIBLE_PROCESS`, `%NEXT_PROCESS`, and `%PREVIOUS_PROCESS` are useful in control structures (`IF`, `WHILE`, `REPEAT`, and so on) and in command procedures.

E.2.2 Monitoring Process Activation and Termination

By default, a tracepoint is triggered when a process comes under debugger control and when it performs an image exit. These predefined tracepoints are equivalent to those resulting from entering the SET TRACE/ACTIVATING and SET TRACE/TERMINATING commands, respectively. You can set breakpoints on these events by using the SET BREAK/ACTIVATING and SET BREAK/TERMINATING commands.

To cancel the predefined tracepoints, use the CANCEL TRACE/PREDEFINED command with the /ACTIVATING and /TERMINATING qualifiers. To cancel any user-defined activation and termination breakpoints, use the CANCEL BREAK command with the /ACTIVATING and /TERMINATING qualifiers (the /USER qualifier is assumed by default when canceling breakpoints or tracepoints).

The debugger prompt is displayed when the first process comes under debugger control. This enables you to enter commands before the main image has started execution, just as with a one-process program.

Also, the debugger prompt is displayed when the last process performs an image exit. This enables you to enter commands after the program has completed execution, just as with a one-process program.

E.2.3 Interrupting the Execution of an Image to Connect It to the Debugger

You can interrupt an image that can be debugged by running it without debugger control in a process and connect it to the debugger.

There are two general scenarios, as follows:

- To start a new debugging session, use the CTRL/Y-DEBUG sequence from DCL level.
- To interrupt the image and connect it to an existing debugging session, use the CONNECT command.

E.2.3.1 Using the CTRL/Y-DEBUG Sequence to Invoke the Debugger

Use the CTRL/Y-DEBUG sequence with the multiprocess debugging configuration exactly as with the default configuration. That is, run the image from DCL level with the RUN/NODEBUG command, then press CTRL/Y to interrupt the image. The DEBUG command invokes the debugger.

The following example shows how you might start a new debugging session:

```
$ DEFINE/JOB DBG$PROCESS MULTIPROCESS
$ RUN/NODEBUG PROG2
.
.
.
CTRL/Y
Interrupt
$ DEBUG
VAX DEBUG Version X5.0-3 MP

%DEBUG-I-INITIAL, language is C, module set to SUB4
trace on activation at SUB4\%LINE 12 in %PROCESS_NUMBER 1
DBG_1>
```

In this example, the DEFINE/JOB command establishes a multiprocess debugging configuration. The RUN/NODEBUG command starts the execution of image PROG2 without debugger control. The CTRL/Y-DEBUG sequence interrupts execution and invokes the debugger.

The VAX DEBUG banner indicates that a new debugging session has been started. The process-specific prompt (DBG_1>) indicates that this is a multiprocess configuration and that execution is suspended in process 1, which is running image PROG2.

The activation tracepoint identifies the location at which execution was interrupted (and, at which point, the debugger took control of the process). You can also use the SHOW CALLS command to display the call stack at that location.

After invoking the debugger, you can use the CONNECT command to bring other processes under debugger control. In the previous example, you could use the CONNECT command to bring processes under debugger control that were created by PROG2 before you interrupted its execution (see Section E.2.3.2).

When using the CTRL/Y-DEBUG sequence, if a multiprocess debugging session already exists in the same job tree as the image that is interrupted, the image connects to that particular session. In this case, because a new session is not started, the VAX DEBUG banner is not displayed when the debugger takes control. This situation could occur if, for example, you entered a SPAWN/NOWAIT command from the session, started execution

with a RUN/NODEBUG command, and then entered a CTRL/Y-DEBUG sequence.

E.2.3.2 Using the CONNECT Command to Interrupt an Image

The CONNECT command, used without a parameter, was introduced in Section E.1.5. (As noted in that section, the CONNECT command is not used to debug VAX C parallel loops.)

When used with a parameter, the CONNECT command enables you to interrupt an image that can be debugged that is running without debugger control and bring it under control of your current debugging session.

The image may have been activated as follows:

- By your program issuing a LIB\$SPAWN run-time library call or a \$CREPRC system service call to spawn a process and run an image without debugger control
- By starting execution with a RUN/NODEBUG command entered at DCL level

In the following example, the CONNECT command causes the image running in process JONES_3 to be interrupted and to come under control of the current debugging session. Process JONES_3 must be in the same job tree as the session.

```
DBG_1> CONNECT JONES_3
```

A process is not identified by a debugger process number until it is connected to a debugging session. Therefore, when specifying a process with the CONNECT command, you can use only its VMS process name or VMS process identification number (PID).

The effect of the CONNECT command is equivalent to attaching to a process from a debugging session and then entering the sequence CTRL/Y-DEBUG to interrupt the running image and invoke the debugger. However, the CONNECT command is simpler for you to enter and also enables you to interrupt a process to which you cannot attach.

E.2.4 Screen Mode Features for Multiprocess Debugging

Screen mode displays, whether predefined or user defined, are associated with the visible process, by default. For example, SRC shows the source code where execution is suspended in the visible process, OUT shows the output of commands executed in the context of the visible process, and so on.

By using the `/PROCESS` qualifier with the `SET DISPLAY` and `DISPLAY` commands, you can create process-specific displays or make existing displays process-specific, respectively. The contents of a process-specific display are generated and modified in the context of that process. You can make any display process specific except for the `PROMPT` display. For example, the following command creates the automatically updated source display `SRC_3`, which shows the source code where execution is suspended in process 3:

```
DBG_2> SET DISPLAY/PROCESS=(%PROC 3) SRC_3 -
_DBG_2> AT RS23 SOURCE (EXAM/SOURCE .%SOURCE_SCOPE\%PC)
```

You assign attributes to process-specific displays in the same way you assign them to displays that are not process specific. For example, the following command makes display `SRC_3` the current scrolling and source display — that is, it causes the output of `SCROLL`, `TYPE`, and `EXAMINE/SOURCE` commands to be directed at `SRC_3`:

```
DBG_2> SELECT/SCROLL/SOURCE SRC_3
```

If you enter a `DISPLAY/PROCESS` or `SET DISPLAY/PROCESS` command without specifying a process, the specified display is then specific to the process that was the visible process when you entered the command. For example, the following command makes `OUT_X` specific to process 2:

```
DBG_2> DISPLAY/PROCESS OUT_X
```

The `/SUFFIX` qualifier appends a process-identifying suffix that denotes the visible process to a display name. This qualifier can be used directly after a display name in any command that specifies a display (for example, `SET DISPLAY`, `EXTRACT`, `SAVE`). It is especially useful within command procedures, in conjunction with display definitions or with key definitions that are bound to display definitions.

In a multiprocess configuration, the predefined tracepoint on process activation automatically creates a new source display and a new instruction display for each new process that comes under debugger control. The displays have the names `SRC_n` and `INST_n`, respectively, where *n* is the process number. These processes are initially marked as removed. They are automatically canceled by the predefined tracepoint on process termination.

Several predefined keypad key sequences enable you to configure your screen with the process-specific source and instruction displays that are automatically created when a process is activated. Table E-3 identifies the keypad keys and describes their general effects. The table also describes any changes to the keypad keys from previous versions of the debugger. Use the `SHOW KEY` command to determine the exact commands issued by these key combinations.

Table E-3: Changed and New Keypad Key Functions

Key	State	Command Invoked or Function
COMMA	GOLD	SELECT/SOURCE %NEXT_SOURCE. Selects the next source display in the display list as the current source display. This function was previously assigned to KP3 in the BLUE state.
KP9	GOLD	SET PROCESS/VISIBLE %NEXT_PROCESS. Makes the next process in the process list the visible process.
KP9	BLUE	Displays two predefined process-specific source displays, SRC_n. These are located at Q1 and Q2, respectively, for the visible process and for the next process on the process list.
KP7	BLUE	Displays two sets of predefined process-specific source and instruction displays, SRC_n and INST_n. These consist of source and instruction displays for the visible process at Q1 and RQ1, respectively, and source and instruction displays for the next process on the process list at Q2 and RQ2, respectively.
KP3	BLUE	Displays three predefined process-specific source displays, SRC_n. These are located at S1, S2, and S3, respectively, for the previous, current (visible), and next process on the process list.
KP1	BLUE	Displays three sets of predefined process-specific source and instruction displays, SRC_n and INST_n. These consist of source and instruction displays for the visible process at S2 and RS2, respectively; source and instruction displays for the previous process on the process list at S1 and RS1, respectively; and source and instruction displays for the next process on the process list at S3 and RS3, respectively.

E.2.5 Setting Watchpoints in Global Sections

You can set watchpoints in global sections. A global section is a region of virtual memory that is shared among all processes of a multiprocess program. A watchpoint that is set on a location in a global section—a global section watchpoint—triggers when any process modifies the contents of that location.

Note that, when setting watchpoints on arrays or structures, performance is improved if you specify individual elements rather than the entire structure with the SET WATCH command.

If you set a watchpoint on a location that is not yet mapped to a global section, the watchpoint is treated as a conventional static watchpoint. For example:

```
DBG_1> SET WATCH ARR[1]
DBG_1> SHOW WATCH
watchpoint of PPL3\ARR[1]
```

When ARR is subsequently mapped to a global section, the watchpoint is automatically treated as a global section watchpoint and an informational message is issued. For example:

```
DBG_1> GO
%DEBUG-I-WATVARNOWGBL, watched variable PPL3\ARR[1] has been remapped
to a global section
predefined trace on activation at routine PPL3 in %PROCESS_NUMBER 2
1:      main()
predefined trace on activation at routine PPL3 in %PROCESS_NUMBER 3
1:      main()
watch of PPL3\ARR[1] at PPL3\%LINE 93 in %PROCESS_NUMBER 2
93:      arr[1] = index;
old value: 0
new value: 1
break at PPL3\%LINE 94 in %PROCESS_NUMBER 2
94:      arr[i] = i;
```

Once the watched location is mapped to a global section, the watchpoint is visible from each process. For example:

```
DBG_2> DO (SHOW WATCH)
For %PROCESS_NUMBER 1
  watchpoint of PPL3\ARR[1] [global-section watchpoint]
For %PROCESS_NUMBER 2
  watchpoint of PPL3\ARR[1] [global-section watchpoint]
For %PROCESS_NUMBER 3
  watchpoint of PPL3\ARR[1] [global-section watchpoint]
```

E.2.6 Compatibility of Multiprocess Commands with the Default Configuration

All the commands, qualifiers, and built-in symbols that are provided for multiprocess debugging are also understood in the default debugging configuration and have similar behaviors (where applicable). For example:

- The EXIT command without a specified parameter terminates a debugging session in both configurations.
- A DO command without the /PROCESS qualifier executes the commands specified in all processes.

- In the default configuration, the visible process is the process that runs the entire program. It is identified as process 1 in a SHOW PROCESS display.
- Built-in symbols such as %PROCESS_NUMBER and %VISIBLE_PROCESS are interpreted correctly in the default configuration.

This compatibility is especially useful because it allows command procedures used for multiprocess debugging to also be used for debugging programs that run in only one process.

E.3 Sample Multiprocess Debugging Session

You can introduce a data dependency in a decomposed VAX C program by not correctly gauging the effect of introducing decomposition pragmas. Consider Example E-1, which has a bug in it.

Example E-1: VAX C Program Used for Multiprocess Debugging Session

```

/* This program demonstrates that a * i = a, where a is
   a matrix and i is the identity matrix. */

#define N 4

double x[N][N], y[N][N], z[N][N];
void count();

main()
{
    int i, j, k, temp;
    double m, n;

    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++)
        {
            x[i][j] = (i + 1) * (j + 10);
            z[i][j] = 0.0;
        }

    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++)
            if (i == j)
                y[i][j] = 1;
            else y[i][j] = 0;

```

(continued on next page)

Example E-1 (Cont.): VAX C Program Used for Multiprocess Debugging Session

```
    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++)
        {
#pragma safe_call count
            for (k = 0; k < N; k++)
            {
                count (x[i][k] * y[k][j], i, j);
            }
        }

#pragma sequential_loop
    for (j = 0; j < N; j++)
#pragma sequential_loop
        for (i = 0; i < N; i++)
            printf(" x[%d][%d] = %f  z[%d][%d] = %f \n",
                j,i,x[j][i],j,i,z[j][i]);
}

void count( double x, int i, int j)
{
    int k;
    double result;

    result = z[i][j] + x;

/* The following loop is inserted to change the timing
   to make the bug obvious.  Otherwise, the program only
   fails sometimes.  */

#pragma sequential_loop
    for (k = 1; k < 100; k++)
    {
        z[i][j] = result;
    }
}
```

The error in this program is that the count function is not safe to use if the k loop is decomposed, because it introduces a dependency into the decomposed loop. If the k loop is decomposed, multiple iterations can write to the same element of z at once. You could use the **#pragma safe_call** directive to instruct VAX C to decompose the loop; however, considering the algorithm in this example, the results would be unpredictable.

A loop was added to the count function in this example, to make the error more obvious. Without the loop to slow down the count function, the timing of the execution of this program often generates the correct results by accident. This timing problem is one of the attributes of a multiprocess

program that makes debugging difficult. An error may not be apparent during parallel execution if the results are only tested once or twice.

Here is the output from Example E-1 on one particular test run:

```
x[0][0] = 10.000000  z[0][0] = 10.000000
x[0][1] = 11.000000  z[0][1] = 11.000000
x[0][2] = 12.000000  z[0][2] = 0.000000
x[0][3] = 13.000000  z[0][3] = 13.000000
x[1][0] = 20.000000  z[1][0] = 0.000000
x[1][1] = 22.000000  z[1][1] = 22.000000
x[1][2] = 24.000000  z[1][2] = 0.000000
x[1][3] = 26.000000  z[1][3] = 26.000000
x[2][0] = 30.000000  z[2][0] = 0.000000
x[2][1] = 33.000000  z[2][1] = 33.000000
x[2][2] = 36.000000  z[2][2] = 0.000000
x[2][3] = 39.000000  z[2][3] = 39.000000
x[3][0] = 40.000000  z[3][0] = 0.000000
x[3][1] = 44.000000  z[3][1] = 44.000000
x[3][2] = 48.000000  z[3][2] = 0.000000
x[3][3] = 52.000000  z[3][3] = 0.000000
```

The numbers in the two columns should match if the program is running correctly.

Since there are errors in the test data, you must debug this program. To debug a multiprocess program, first set up a logical name for the multiprocessor debugger. Enter the following command at the DCL (\$) prompt:

```
$ DEFINE/JOB DBG$PROCESS MULTIPROCESS
```

When you start a multiprocessor debugging session, as shown in Example E-2, enter the GO command. This takes you past the initialization of the detached processes that are used to run the decomposed loops in parallel. Since the elements of z are incorrect, set a breakpoint at line 66, the line in which z is set. Using the DO command to set a breakpoint causes the debugger to stop execution when any one of the parallel processes executes line 66.

Example E-2: Sample Multiprocess Debugging Session

VAX DEBUG Version V5.0-00 MP

```
%DEBUG-I-INITIAL, language is C, module set to DEBUGERR
❶ %DEBUG-I-NOTATMAIN, type GO to get to start of main program
❷ predefined trace on activation at FOR$INIT_PARALLEL+36 in %PROCESS_NUMBER 1
DBG_1> GO
break at routine DEBUGERR\main in %PROCESS_NUMBER 1
   26: {
❸ DBG_1> STEP
predefined trace on activation at FOR$INIT_PARALLEL+36 in %PROCESS_NUMBER 3
predefined trace on activation at FOR$INIT_PARALLEL+36 in %PROCESS_NUMBER 2
predefined trace on activation at FOR$INIT_PARALLEL+36 in %PROCESS_NUMBER 4
stepped to DEBUGERR\main\main1\%LINE 35+11 in %PROCESS_NUMBER 1
   35:      }
❹ DBG_1> SHOW PROCESS/ALL
   Number Name           Hold State   Current PC
*  1 Beth                step      DEBUGERR\main\main1\%LINE 35+11
  2 FOR$4E401520_01      interrupted SHARE$DBGSSISHR+9910
  3 FOR$4E401520_02      interrupted SHARE$DBGSSISHR+8074
  4 FOR$4E401520_03      interrupted SHARE$DBGSSISHR+8074
❺ DBG_1> DO (SET BREAK %LINE 66)
❻ DBG_1> GO
break at DEBUGERR\count\count5\%LINE 66 in %PROCESS_NUMBER 2
   66:      result = z[i][j] + x;
break at DEBUGERR\count\count5\%LINE 66 in %PROCESS_NUMBER 1
   66:      result = z[i][j] + x;
❼ DBG_1> SET PROCESS %PROC 2
DBG_2> EX I,J
DEBUGERR\count\i:      0
DEBUGERR\count\j:      0
❽ DBG_2> SET PROCESS %PROC 1
DBG_1> EX I,J
DEBUGERR\count\i:      0
DEBUGERR\count\j:      0
```

-
- ❶ You must enter the GO command to set up the parallel-processing environment before entering other debugger commands.
 - ❷ The debugger automatically traces the activation of FOR\$INIT_PARALLEL, the procedure that is executed when the parallel processing environment is enabled.
 - ❸ Entering the STEP command shows that four processes have been initialized to handle the decomposition of loops.
 - ❹ The SHOW PROCESS/ALL command verifies that the four processes exist.

- ⑤ Setting a break at line 66 causes the debugger to stop at line 66 so that you can examine the problem variables in function count. The DO command is used for setting a breakpoint across all the processes running in the DEBUG session.
- ⑥ After entering the GO command, the debugger stops at line 66, which is being executed by processes 1 and 2 at the same time.
- ⑦ We set the current process to 2 and examine the variables i and j, both of which contain 0.
- ⑧ We set the current process to 1 and examine the same variables, which contain 0.

This example shows that processes 1 and 2 are updating the same element of array z while processing loop k inside function count. The program incorrectly uses the **#pragma safe_call** directive in this example, because function count contains a loop-carried data dependency.

E.4 Considerations for Multiprocess Debugging

Several users debugging programs that occupy several processes can place a significant load on a system. This section describes the resources used by the multiprocess debugger and how to tune your system accordingly.

The following discussion covers only the resources used by the debugger. You may also have to tune your system to support the execution of the multiprocess programs themselves (see Section 3.9).

E.4.1 User Quotas

Each user needs sufficient PRCLM quota to create an additional subprocess for the debugger, beyond the number of processes needed for the multiprocessing program. This quota may need to be increased to account for the debugger subprocess.

BYTLM, ENQLM, FILLM, and PGFLQUO are pooled quotas. These quotas may need to be increased to account for the debugger subprocess, as follows:

- Each user's ENQLM quota should be increased by at least the number of processes being debugged.
- Each user's PGFLQUO quota may need to be increased. If a user has insufficient PGFLQUO, the debugger may fail to activate, or produce "virtual memory exceeded" errors during execution.

- Each user's FILLM and BYTLM quotas may need to be increased. The debugger requires enough FILLM and BYTLM quotas to open each image file being debugged, the corresponding source files, and the DEBUG input, output, and log files. The DEBUG SET MAX_SOURCE_FILES command can be used to limit the number of source files kept open by the debugger at any one time.

E.4.2 System Resources

The kernel and main debugger communicate through global sections. The main debugger communicates with up to 8 kernel debuggers through a 65-page global section. Therefore, the SYSGEN parameters GBLPAGES and GBLSECTIONS may need to be increased. For example, if 10 users are using the debugger simultaneously, 10 global sections (GBLSECTIONS), using a total of 650 global pages (GBLPAGES), are required by the debugger.

VAX C Glossary

additive operator

An operator that performs addition (+) or subtraction (-). These operators perform arithmetic conversion on each of the operands, if necessary. See also *arithmetic conversion rules*.

aggregate

A data structure (array, structure, or union) composed of segments called members. You declare the members to be of either a scalar or aggregate data type. Members of an array are called elements and must be of the same data type. A structure has named members that can be of different data types. A union is a structure that is as long as its longest declared member and that contains the value of only one member at a time.

ampersand (&)

As a unary operator, computes the address of its operand. As a binary operator, performs a bitwise AND on two operands; both must be of integral type. As an assignment operator (&=), performs a bitwise AND on two expressions and assigns the result to the left object. The double ampersand (&&), a binary operator, performs a logical AND on two operands. See also *binary operator*, *bitwise operator*, *logical operator*, and *unary operator*.

argument

An expression that appears within the parentheses of a function call. The expression is evaluated and the result is copied into the corresponding parameter of the called function. See also *argument passing* and *parameter*.

argument passing

The mechanism by which the value of the argument in a function call is copied to a parameter in the called function. In VAX C, all arguments are passed by value; that is, the parameter receives a copy of the argument's value. Therefore, a function called in VAX C cannot modify the value of an argument except by using its address. In general, addresses are passed using the ampersand operator (see *ampersand* (&)) in the function call or by passing a pointer variable. In addition, using an array or function name (an array with no brackets or function identifier with no parentheses) as an argument results in the passing of the address of the array or function.

arithmetic conversion rules

The set of rules that govern the changing of a value of an operand from one data type to another in arithmetic expressions. Conversions take place in assignments by changing the type of the right operand's result to that of the object referred to by the left operand; the resultant type also applies to the assignment expression. Conversions are also performed when arguments are passed to functions.

arithmetic operator

A VAX C operator that performs a mathematical operation. In an expression, certain operations take precedence (are performed first) over other operations. The unary minus operator (–) is at the highest level of precedence. At the next level are the binary operators for multiplication (*), division (/), and mod (%). At the next level are addition (+) and subtraction (–). There is no unary plus operator, and there is no exponentiation operator. If necessary, all the binary operators perform the arithmetic conversions on their operands. See also *arithmetic conversion rules*.

arithmetic type

One of the integral data types, enumerated types, **float**, or **double**.

array

An aggregate data type consisting of subscripted members, called elements, all of the same type. Elements of an array can be one of the fundamental types or can be structures, unions, or other arrays (to form multidimensional arrays).

assignment expression

An expression that has the following form:

E1 asgnop E2

Expression E1 must evaluate to an lvalue, the operator asgnop is an assignment operator, and E2 is an expression. The type of an assignment expression is that of its left operand. The value of an assignment expression is that of the left operand after the assignment takes place. If the operator is of the form op=, then the operation E1 op (E2) is performed, and the result is assigned to the object referred to by E1; E1 is evaluated once.

assignment operator

The combination of an arithmetic or bitwise operator with the assignment symbol (=); also, the assignment symbol by itself. See also *assignment expression*.

asterisk (*)

As a unary operator, treats its operand as an address and results in the contents of that address. As a binary operator, multiplies two operands, performing the arithmetic conversions, if necessary. As an assignment operator (*=), multiplies an expression by the value of the object referred to by the left operand, and assigns the product to that object. See also *unary operator* and *binary operator*.

binary operator

An operator that is placed between two operands. The binary operators include arithmetic operators, shift operators, relational operators, equality operators, bitwise operators (AND, OR, and XOR), logical connectives, and the comma operator, in that order of precedence. All binary operators group from left to right. VAX C has no exponentiation operator. The VAX C RTL function **exp** must be used instead.

bitwise operator

An operator that performs Boolean algebra on the binary values of two operands, which must be integral. If necessary, the operators perform the arithmetic conversions. Both operands are evaluated. All bitwise operators are associative, and expressions using them may be rearranged. The operators include, in order of precedence, the single ampersand (&) (bitwise AND), the circumflex (^) (bitwise exclusive OR), and the single bar (|) (bitwise inclusive OR).

block

See *compound statement*.

block activation

The run-time activation of a block or function, in which local **auto** and **register** variables are allocated storage and, if they are declared with initializers, given initial values. Variables of storage class **static**, **extern**, **globaldef**, and **globalvalue** are allocated and initialized at link time. The block activation precedes the execution of any executable statements in the function or block. Functions are activated when they are called. Internal blocks (compound statements) are activated when the program control flows into them. Internal blocks are not activated if they are entered by a **goto** statement, unless the **goto** target is the label of the block rather than the label of some statement within the block. If a block is entered by a **goto** statement, references to **auto** and **register** variables declared in the block are still valid references, but the variables may not be properly initialized. Blocks that make up the body of a **switch** statement are not activated; **auto** or **register** variables declared in the block are not initialized.

built-in functions

The function definitions that are part of the compiler. A call to one of these functions does not call a function in a run-time library or in your program. Most of the built-in functions access the VAX hardware instructions to perform operations quickly that are cumbersome, slow, or impossible in the VAX C language.

cast

An expression preceded by a cast operator of the form (type_name). The cast operator forces the conversion of the evaluated expression to the given type. The expression is assigned to a variable of the specified type, which is then used in place of the whole construction. The cast operator has the same precedence as the other unary operators.

character

- A member of the ASCII character set.
- An object of the VAX C data type **char**, which is stored in a single byte of memory. An object of type **char** always represents a single character, not a string.
- A constant of type **char**, consisting of up to four ASCII characters enclosed in apostrophes (' ') not quotation marks (" ").

See also *string*.

comma operator

A VAX C operator used to separate two expressions as follows:

E1, E2

The expressions E1 and E2 are evaluated left to right, and the value of E1 is discarded. The type and value of the comma expression are those of E2.

comment

A sequence of characters introduced by the pair `(/*`) and terminated by `(*/`). Comments are ignored during compilation. They may not be nested.

Common Data Dictionary (CDD)

An optional VMS software product, available under a separate license, that maintains a set of data structure definitions that many programs on a system, written in many languages, can access. The language-independent definitions are translated into the target language when they are included in the program stream. You can include the CDD records in VAX C programs using the **#dictionary** preprocessor directive. The **#dictionary** directive is VAX C specific and is not portable.

compilation unit

All the source files compiled to form a single object module. In other C documentation, the term source file is synonymous with the VMS compilation unit, which is not necessarily a single source file. Declarations and definitions within a compilation unit determine the lexical scope of functions and variables.

compound statement

Valid VAX C statements enclosed in braces `{ }`. Compound statements can also include declarations. The scope of these variables is local to the compound statement. A compound statement, when it is not the body of a function, is called a block.

conditional operator

The VAX C operator `(?:)`, which is used in conditional expressions of the following form:

E1 ? E2 : E3

E1, E2, and E3 are valid VAX C expressions. E1 is evaluated, and if it is nonzero, the result is the value of E2; otherwise, the result is the value of E3. Either E2 or E3 is evaluated, but not both.

constant

A primary expression whose value does not change. A constant may be literal or symbolic.

control dependency

A type of data dependency that inhibits parallel processing of a program. A control dependency is an inhibiting factor that involves the flow of control in a program (for instance, by using the **goto** statement). VAX C does not check programs that are compiled with the **/PARALLEL** qualifier for control dependencies, since the compiler's parallel-processing support only includes loop decomposition. See also *data dependency*.

constant expression

An expression involving only constants. Constant expressions are evaluated at compile time so they may be used wherever a constant is valid.

conversion

The changing of a value from one data type to another. Conversions take place in assignments by changing the type of the right operand's result to that of the object referred to by the left operand; the resultant type also applies to the assignment expression. Conversions are also performed when arguments are passed to functions: **char** and **short** become **int**; **unsigned char** and **unsigned short** become **unsigned int** if no function prototype is in scope; **float** becomes **double**. Conversions can also be forced by means of a cast. Conversions are performed on operands in arithmetic expressions by the arithmetic conversions.

conversion characters

A character used with the VAX C RTL Standard I/O functions that is preceded by a percent sign (%) and specifies an input or output format. For example, letter d instructs the function to input/output the value in a decimal format.

Curses

A screen management package comprised of VAX C RTL functions and macros that create and modify defined sections of the terminal screen, and optimize cursor movement. Curses defines rectangular regions on the terminal display that you may write upon, rearrange, move to new positions on the screen, and delete from the screen. These rectangular regions are called windows. To use any of the Curses functions or macros, you must include the *curses* definition module with the **#include** preprocessor directive.

data definition

The syntax that both declares the data type of an object and reserves its storage. For variables that are internal to a function, the data definition is the same as the declaration. For external variables, the data definition is external to any function (an external data definition).

data-type modifier

Keywords that affect the allocation or access of data storage. The two data-type modifiers are **const** and **volatile**.

data dependency

Lines of code that depend on sequential execution for correct and predictable memory access. If a program contains data dependencies, it is not a good candidate for parallel processing. By default (when the /PARALLEL qualifier is used on the CC command line), VAX C checks all **for** and some **while** loops for occurrences of function calls, pointer references, array elements that are modified and accessed by two or more iterations of the loop, and references to scalar variables. If the loop contains such an occurrence, VAX C does not run the loop in parallel due to possible data dependencies. See also *parallel processing*.

declaration

A statement that gives the data type and possibly the storage class of one or more variables.

decomposition

A division of a loop into groups of iterations that execute in separate subprocesses on separate processors. The order of the execution of loop iterations is not guaranteed. See also *parallel processing*.

DEC/Shell

An optional VMS software product available under a separate license that is a command-language interpreter based on the UNIX Version 7.0 Bourne Shell with commands for interactive program development, device and data file manipulation, and interactive and batch execution. DEC/Shell RTL functions were added to the VAX C RTL so that valid DEC/Shell file specifications could be used in VAX C source programs. See also *file specification*.

dictionaries

A hierarchical organization, similar to the organization of directories and subdirectories, of data structure definitions in the Common Data Dictionary (CDD). See also *Common Data Dictionary (CDD)*.

directives

See *preprocessor directives*.

elements

Members of an array. See also *aggregate*.

enumerated type

A type defined (with the **enum** keyword) to have an ordered set of integer values. The integer values are associated with constant identifiers named in the declaration. Although **enum** variables are stored internally as integers, use them in programs as if they have a distinct data type named in the **enum** declaration.

equality operator

One of the operators equal to (==) or not equal to (!=). They are similar to the relational operators, but at the next lower level of precedence.

exponentiation operator

The VAX C language does not have an exponentiation operator. Use the VAX C RTL **exp** function.

expression

A series of characters that the compiler can use to produce a value. Expressions have one or more operands and, usually, one or more operators. An identifier with no operator is an expression that yields a value directly. Operands are either identifiers (such as variable names) or other expressions, which are sometimes called subexpressions. See also *operator* and *macros*.

external storage class

A storage class that permits identifiers to have a link-time scope that can possibly span object modules. Identifiers of this storage class are defined outside of functions using no storage-class specifier, and are declared, optionally, throughout the program using the **extern** specifier. External variables provide a means other than argument passing for exchanging data between the functions that comprise a VAX C program. See also *link-time scope*.

file descriptor

In the UNIX environment, the integer that identifies a file. The VMS equivalent is a file pointer.

file specification

An identifier that specifies an existing file. There are two types of valid file specifications in VAX C: VMS specifications and DEC/Shell specifications. DEC/Shell specifications are a subset of UNIX specifications.

floating type

One of the data types **float** or **double**, representing a single- or double-precision, floating-point number. There are two implementations of the data type **double**: **D_floating** and **G_floating**. The range of values for the **D_floating** variables is the same as that for **float** variables, but the precision is 16 decimal digits, as opposed to 7. Programs that use **G_floating** variables must use the **/G_FLOAT** command-line qualifier. A **G_floating** variable has considerably greater range, but has less precision.

function

The primary unit from which VAX C programs are constructed. A function definition begins with a name and parameter list, followed by the declarations of the parameters (if any) and the body of the function enclosed in braces ({}). The function body consists of the declarations of any local variables and the set of statements that perform its action. Functions do not have to return a value to the caller. All VAX C functions are external; that is, a function may not contain another function. See also *function call*.

function call

A primary expression, usually a function identifier followed by parentheses, that is used to invoke the function. The parentheses contain a (possibly empty) comma-separated list of expressions that are the arguments to the function. Any previously undeclared identifier followed immediately by parentheses is declared as a function returning **int**. Any function may call itself recursively.

function inline expansion

A replacement of a function call with code that performs the actions of the defined function. This process reduces execution time. By default, VAX C attempts to expand inline all functions. You can use the **#pragma inline** directive to provide inline expansion for functions that VAX C does not expand inline by default. See also *pragma*.

function unrolling

See *function inline expansion*.

fundamental type

The set of arithmetic data types plus pointers. In general, the fundamental types comprise those data types that can be represented naturally on a VAX; usually, this means integers and floating-point numbers of various machine-dependent sizes, and machine addresses.

global storage class

A storage class that permits identifiers to have a link-time scope that can possibly span object modules. Identifiers of this storage class are defined using the **globaldef** storage-class specifier, and are declared, optionally, throughout the program using the **globalref** specifier. You can use the **globalvalue** specifier to define a global symbol, or constant. Global variables provide a means other than argument passing for exchanging data between the functions that comprise a VAX C program. See also *link-time scope*.

identifier

A sequence of letters and digits, the first 255 of which must be unique. The underscore (`_`) and dollar sign (`$`) are letters in this context. The first character of an identifier must be a letter. Upper- and lowercase letters specify different identifiers in VAX C. However, all external names are converted to uppercase to be consistent with the VMS environment and are only 31 characters in length.

initializer

The part of a declaration that gives the initial value(s) for the preceding declarator. An initializer consists of an equal sign (`=`) followed by either a single expression or a comma-separated list of one or more expressions in braces.

inline expansion

See *function inline expansion*.

integral type

One of the data types **char** or **int** (all sizes, signed or unsigned).

internal storage class

A storage class that permits identifiers declared inside of a function body to be recognized only from the declaration to the end of the immediately enclosing block. Identifiers of the internal storage class are declared using the **auto** and **register** storage-class specifiers. See also *scope*.

keyword

A character string that is reserved by the VAX C language and cannot be used as an identifier. Keywords identify statements, storage classes, data types, and the like. VAX C RTL function names are not VAX C keywords; you may redefine function names.

lexical scope

The area in which the compiler recognizes a declared identifier within a given compilation unit. See also *scope*.

License Management Facility (LMF)

A process by which you register and use some DIGITAL software products. See the *VAX C Installation Guide* for more information.

lifetime

The length of time for which storage for a variable is allocated. See also *program section* (psect), *internal storage class*, and *external storage class*.

link libraries

The libraries searched by the VMS Linker to resolve external references. Depending on the needs of your program, you have to specify certain libraries in a specific order so that your program links properly. For more information, see Chapter 1.

link-time scope

The area in which the VMS Linker recognizes an identifier within a given program. See also *scope*.

literal

A constant whose value is written explicitly in the program. Literal values have type **int** or **double**, depending on their forms. Character constants have type **int**. Floating constants have type **double**. Character-string constants have type array of **char**.

local variable

A variable declared inside a function body. See also *internal storage class*.

logical expression

An expression made up of two or more operands separated by a logical operator. Each operand must be a fundamental type or must be a pointer or other address expression. Operands do not have to be the same type. Logical expressions always return 1 or 0 (type **int**) to indicate a true or false value, respectively. Logical expressions are always evaluated from left to right, and the evaluation stops as soon as the result is known.

logical operator

One of the binary operators logical AND (**&&**) and logical OR (**||**).

loop

A construct that executes a single statement or a block repeatedly until a given expression evaluates to false. The single statement or block is called the loop body. VAX C has three types of loops: one that evaluates the expression before executing the loop body (the **while** statement), one that evaluates the expression after executing the loop body (the **do** statement), and one that executes the loop body a specified number of times (the **for** statement).

loop decomposition

See *decomposition*.

loop-carried dependency

A type of data dependency that inhibits parallel processing of a program. A loop-carried dependency is an inhibiting factor that involves function calls, array access, pointer references, and scalar references inside of a loop that is being considered for decomposition. VAX C checks programs for loop-carried dependencies. See also *data dependency*.

loop-independent dependency

A type of data dependency that inhibits parallel processing of a program. A loop-independent dependency is an inhibiting factor that involves the relative position of two statements in the program that are outside of all loops. VAX C does not check programs for loop-independent dependencies, since the compiler's parallel-processing support only includes loop decomposition. See also *data dependency*.

lvalue

The address in memory that is the location of an object whose contents can be assigned or modified. In this guide, the term describes a category in VAX C grammar. An expression evaluating to an lvalue is required on the left side of an assignment operator (hence its name) and as the operand of certain other operators, such as the increment (++) and decrement (--) operators. A variable name is an example of an expression evaluating to an lvalue, since its address can be taken (with &), and values can be assigned to it. A constant is an example of an expression that is not an lvalue. See also *rvalue*.

macro

A text substitution that is defined with the **#define** preprocessor directive and can include a list of parameters. The parameters in the **#define** directive are replaced at compile time with the corresponding arguments from a macro reference encountered in the source text.

main_program option

A tag that can be placed on a separate line between the function parameter list and the rest of a function definition to tell the VMS image activator to begin program execution with this function. You can use the identifier `main_program` when there is no function named `main`; it is not a keyword; it can be spelled in upper- or lowercase; and it is VAX C specific.

members

Segments of the aggregate data structures (arrays, structures, or unions) that are declared to be of either scalar or aggregate data type. See also *aggregate*.

module

- The object code produced and placed into a file with a `.OBJ` extension after a compilation unit has been compiled. The object file is the file name with the `.OBJ` extension; the object module is the system-recognized name (usually the same as the object-file name without an extension).
- A segment of object code located in an object library.

multiplication operator

An operator that performs multiplication (*), division (/), or modular arithmetic (%). If necessary, it performs the arithmetic conversions on its operands. The mod operator (%) yields the remainder of the first operand divided by the second.

null pointer

A pointer variable that has not been assigned an lvalue and whose value has been initialized to zero. If you use a null pointer in an expression that needs a value, VAX C will let you try to access memory location zero, which will cause the ACCVIO hardware error.

NUL character

The escape sequence (\0) that VAX C uses to terminate all character strings.

object

Data stored at a location in memory represented by an identifier. Objects are one of the basic elements that the language can manipulate; that is, the elements to which operators can be applied. In VAX C, objects include data (such as integers, real numbers, or characters), data structures (arrays, structures, or unions), and functions.

occlude

In the Curses Screen Management package, when the area of one defined window overlaps the area of another defined window on the terminal screen. See also *Curses*.

operator

A character that performs an operation on one or more operands. In order of precedence (high to low), operators are classified as the primary-expression operators, unary operators, binary operators, the conditional operator, assignment operators, and the comma operator.

parallel processing

A process by which a program's lines of code are divided into groups that are run concurrently on several processes (each processor runs separate subprocesses). See also *decomposition*.

parameter

A variable listed in the parentheses and declared between the function identifier and body in the function definition. The parameter receives a copy of the value of an associated argument when the function is called. The items in parentheses in a macro definition are also called parameters, but the semantics are different from VAX C function calls.

pointer

A variable that contains the address (lvalue) of another variable or function. A pointer is declared with the unary asterisk operator (*).

portability

The ability to compile an unaltered C source program on several operating systems and machines; in this guide particularly, between UNIX and VMS systems.

pragma

A preprocessor directive that produces implementation-specific results. Certain pragmas may not be portable, but other compilers may support pragmas that are supported by VAX C. See also *preprocessor directives*.

precedence of operators

The order in which operations are performed. If an expression contains several operators, the operations are executed in the following order: primary expression operators, unary operators, binary operators, the conditional operator, assignment operators, and the comma operator.

preprocessor directives

Lines of text in a VAX C source file that change the order or manner of subsequent compilation. The directives are **#define**, for macro substitution and other replacements; **#undef**, to cancel a previous **#define**; **#include**, to include an external source text; **#line**, to specify a line number to the compiler; **#module**, to specify a module name to the linker; **#dictionary**, to extract data structures from the Common Data Dictionary; **#pragma** to give the compiler implementation-specific information; and **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, **#endif**, to place conditions on the compilation of sections of a program. In VAX C, these directives are processed by an early phase of the compiler, not by a separate program.

primary expression

An expression that contains only a primary-expression operator, or no operator. Primary expressions include previously declared identifiers, constants, strings, function calls, subscripted expressions, and references to structure or union members.

primary-expression operator

A VAX C operator that qualifies a primary expression. The set of such operators consists of paired brackets ([]) to enclose a single subscript; paired parentheses (()) to enclose an argument list or to change the associative precedence of operators; a period (.) to qualify a structure or union name with the name of a member; and an arrow (->) to qualify a structure or union member with a pointer or other address-valued expression.

program section (psect)

An area of virtual memory that has a name, a size, and a series of attributes that describe the intended or permitted usage of that permanent variable. Variables of type **static**, and of all external and global types are placed in psects. See also *lifetime*.

refresh

A Curses Screen Management term describing the updating of the terminal screen so that the latest contents of defined windows are placed on the screen. No edits made to any window can appear on the terminal screen until you refresh the window on the screen using **refresh**, **wrefresh**, or **touchwin**. See also *Curses*.

relational operator

One of the operators less than (<), greater than (>), less than or equal to (<=), or greater than or equal to (>=). The result (which is of type **int**) is 1 or 0, indicating a true or false relation, respectively. If necessary, the arithmetic conversions are performed on the two operands. Relational operators group from left to right.

run-time library

In VAX C, the group of common functions and macros that accompany the compiler that may be called to perform I/O tasks, character-string manipulation, math tasks, system calls, and various other tasks. The C language includes no facilities to administer I/O, so compilers include run-time libraries to provide this service. The VAX C RTL is shipped with the VMS operating system. You can access the VAX C RTL by receiving a copy of the function module in your program's image, or by sharing the function image with your program so that control is passed

to the function image and then back to your program. See also *shareable image*.

rvalue

The object stored at a location in memory represented by an identifier. The rvalue of a variable is the variable's object. See also *lvalue* and *object*.

scalar

Single objects, including pointers, that can be manipulated in their entirety, in an arithmetic expression. See also *object* and *aggregate*.

scope

The portion of a program in which a particular name has meaning. The link-time scope of names declared in external definitions possibly extends from the point of the definition's occurrence to the end of the program. The scope of the names of function parameters is the function itself. The scope of names declared in any block (that is, after the brace beginning any compound statement) is restricted to that block. Names declared in a block supersede any other declaration of the name, including external definitions, for the extent of that block. Tags within **struct**, **union**, **typedef**, and **enum** declarations are identifiers that are subject to the same scope rules as any identifiers. Member names in structure or union references are not subject to the same scope rules (see *uniqueness*). The scope of a label is the entire function containing the label.

shareable image

A VMS image that passes control to another image that passes control back to the original program. You can access the VAX C RTL as a shared image; control is passed to the VAX C RTL and then back to your program instead of a copy of the function's object module being copied into your program's image.

shift operator

One of the binary operators (<<) or (>>). Both operands must have integral types. The value of the expression $E1 \ll E2$ is the result of expression $E1$ (interpreted as a bit pattern) left-shifted by $E2$ bits. The value of $E1 \gg E2$ is $E1$ right-shifted by $E2$ bits.

statement

The language elements that perform the action of a function. Statements include expression statements (an expression followed by a semicolon), null statements (the semicolon by itself), compound statements (blocks), and an assortment of statements identified by keywords (such as **return**, **switch**, and **do**).

static storage class

A storage class that permits identifiers to be recognized possibly from the point of the declaration to the end of the compilation unit. Identifiers of the static storage class are declared using the **static** storage-class specifier. See also *scope*.

stderr

The predefined file pointer associated with the terminal to report run-time errors. The pointed file is equivalent to the VMS logical SYS\$ERROR and the file descriptor 2. To use this definition, include the *stdio* definition module in your source code using the **#include** preprocessor directive.

stdin

The predefined file pointer associated with the terminal to perform input. The pointed file is equivalent to the VMS logical SYS\$INPUT and the file descriptor 0. For example, if you specify *stdin* as the pointer to the file to read from in the **getc** macro, the macro reads from the terminal. To use this definition, include the *stdio* definition module in your source code using the **#include** preprocessor directive.

stdout

The predefined file pointer associated with the terminal to perform output. The pointed file is equivalent to the VMS logical SYS\$OUTPUT and the file descriptor 1. For example, if you specify *stdout* as the pointer to the file to write to in the **putc** macro, the macro writes to the terminal. To use this definition, include the definition module *stdio* in your source code using the **#include** preprocessor directive.

storage class

The attribute that, with its type, determines the location, lifetime, and scope of an identifier's storage. Examples are **static**, **external**, and **auto**.

storage-class modifier

Keywords used with the storage-class and data-type keywords to change program section attributes of variables, which restricts access to them. The two storage-class modifiers are **noshare** and **readonly**.

string

- An array of type **char**.
- A constant consisting of a series of ASCII characters enclosed in quotation marks. Such a constant is declared implicitly as an array of **char**, initialized with the given characters, and terminated by a NUL character (ASCII 0, VAX C escape sequence `\0`).

structure

An aggregate type consisting of a sequence of named members. Each member may have either a scalar or an aggregate type. A structure member may also consist of a specified number of bits, called a field.

symbolic constant

An identifier assigned a constant value by a **#define** directive. You may use a symbolic constant wherever a literal is valid.

tags

Identifiers that represent a declaration of the data types **struct**, **union**, or **enum**. You may use tags in declarations from that point onward in the program to declare other variables of the same type without having to key in the lengthy declaration again.

tokens

The fundamental elements making up the text of a VAX C program. Tokens are identifiers, keywords, constants, strings, operators, and other separators. White space (such as spaces, tabs, newlines, and comments) is ignored except where it is necessary to separate tokens.

type

The attribute that, with its storage class, determines the meaning of the values found in the identifier's storage. Types include the integral and floating types, pointers, enumerated types, the **void** data type, and the derived types array, function, structure, and **union**.

type name

The declaration of an object of a given type that omits the object identifier. A type name is used as the operand of the cast and **sizeof** operators.

unary operator

An operator that takes a single operand. In VAX C, some unary operators can be either prefixed or placed after the operand. The set includes the asterisk (indirection), ampersand (address of), minus (arithmetic unary minus), exclamation (logical negation), tilde (one's complement), double plus (increment), double minus (decrement), cast (force type conversion), and **sizeof** (yields the size, in bytes, of its operand) operators.

union

A union is an aggregate type that can be considered a structure, all of whose members begin at offset 0 from the base, and whose size is sufficient to contain any of its members. A union can only contain the value of one member at a time.

uniqueness

A property of the names used for certain structure and union members. A name is unique if either of the following conditions is true:

- The name is used only once
- The name is used in two or more different structures (or unions), but each use denotes a member at the same offset from the base and of the same data type

The significance of uniqueness is that a unique member name can possibly be used to refer to a structure in which the member name was not declared (although a warning message is issued).

variable

An identifier used as the name of an object.

value

The result of an expression. For example, when a variable on the right side of an assignment expression is evaluated, the value obtained is the object (rvalue) of the variable; when a variable on the left side of an assignment expression is evaluated, the value obtained is the address (lvalue) of the variable.

white space

Spaces, tabs, newlines, and comments. VAX C defines where you can and cannot place these characters.

windows

In the Curses Screen Management package, the defined rectangular regions on the terminal screen that you can write upon, rearrange, move to new positions on the screen, and delete from the screen. You define windows by specifying the upper left corner coordinate, the number of lines, and the number of columns comprising the window. To see the results after editing a window, you must refresh the window on the terminal screen. See also *refresh*.

!(logical expression), 7–10
!= (inequality operator), 7–16
% (modulus operator), 7–15
& (address operator), 7–11
& (bitwise AND operator), 7–17
&& (logical AND operator), 7–17
() (parenthetical expressions), 7–3
() (cast operator), 7–13
* (indirection operator), 7–11
* (multiplication operator), 7–15
+ (addition operator), 7–15
++ (increment operator), 7–10
, (comma operator), 7–22
. (structure and union operator), 7–6
/ (division operator), 7–15
\0 (NUL character), 8–8
== (equality operator), 7–16
[] (bracket operators), 7–5
< (less-than operator), 7–16
<< (left shift operator), 7–19
<= (less-than or equal-to operator), 7–16
= (assignment operator), 7–20
+= (assignment operator), 7–20
-= (assignment operator), 7–20
*= (assignment operator), 7–20
-> (structure or union pointer operator), 7–6
> (greater-than operator), 7–16
>= (greater-than or equal-to operator), 7–16
>> (right shift operator), 7–19
?: (conditional operator), 7–19
- (subtraction operator), 7–10
-- (decrement operator), 7–10
^ (bitwise XOR operator), 7–17
| (bitwise OR operator), 7–17
|| (logical OR operator), 7–17

A

Abort function, 2–6
Access mode
 record, 12–4
ACCVIO
 hardware error, 8–11
/ACTIVATING qualifier (DEBUG MP)
 SET TRACE command, E–15
Activation (DEBUG MP)
 predefined tracepoint, multiprocess program, E–15
_ADAWI built-in function, 11–5
Additive operators, 7–15
Address expression (DEBUG)
 with DEPOSIT command, 2–21
 with EVALUATE command, 2–21
 with EXAMINE command, 2–20
 with SET BREAK command, 2–15
 with SET TRACE command, 2–17
 with SET WATCH command, 2–18
Address-of operator (&), 7–11
 restrictions, 8–30
 tutorial information, 4–18
 used with pointers, 8–12
Aggregates, 8–15 to 8–32
 arrays, 8–15
 See also, Bracket operators ([])
 debugger access to, 2–28
 defined, 8–2, 8–15
 struct, 8–20
 structures, 8–15
 tutorial information, 4–21
 unions, 8–15, 8–20
 variant, 8–28
_align storage-class modifier, 9–25
/ANALYSIS_DATA CC qualifier, 1–8

- AND bitwise operator (&), 7–17
- ANSI standard, 4–2
- argc main function argument, 5–15
- Arguments, 5–12 to 5–13
 - Command-line (DCL), 5–15
 - conversion of function arguments, 5–13, 7–3
 - function and array identifiers as, 5–13
 - function prototypes, 5–9
 - functions used as, 5–14
 - in **#define** preprocessor macros, 10–4
 - list of
 - variable-length, 13–52
 - null in a system routine, 13–45
 - optional for system routines, 13–49
 - passing
 - by descriptor, 13–7, 13–14
 - by immediate value, 13–7, 13–8
 - floating-point values, 13–10
 - by reference, 13–7, 13–11
 - by value, to a VAX C function, 5–12
 - passing mechanisms in mixed-language programming, 13–6
 - rules governing, 5–12
 - specified to the main function, 5–15
 - to a function
 - conversion of, 7–24
 - tutorial information, 4–5
- argv main function argument, 5–15
- Arithmetic conversion, 7–22
 - summary of, D–7
- Arrays, 8–15
 - as expressions, 7–5
 - data dependencies introduced during parallel processing, 3–11
 - debugger access to, 2–25
 - declaration of, 8–15
 - initializing, 8–18
 - references to, 7–5
 - tutorial information, 4–21
- /ASCII qualifier, 2–27
- ASCII
 - byte values, 8–8
 - NUL character, 8–8
 - in character constants, 8–6
- Assignment operators, 7–8 to 7–9
 - precedence of, 7–8
- Asterisk operator (*), 8–11
- AUTHORIZE Utility
 - tuning for parallel processing, 3–36
- [auto]** keyword, 9–10 to 9–11
 - affect on parallel processing, 3–13

- [auto]** keyword (cont'd.)
 - scope of, 9–5
 - used in declarations inside of blocks, 6–3
 - with scalar variables during parallel processing, 3–17

B

- \b (backspace), 8–7
- _BBCCI built-in function, 11–6
- _BBSSI built-in function, 11–6
- Binary operators
 - additive, 7–15
 - bitwise, 7–17
 - equality, 7–16
 - logical, 7–17
 - multiplication, 7–15
 - precedence of, 7–8
 - relational, 7–16
 - shift, 7–19
- Bit fields, 8–30 to 8–32
- Bitwise operators, 7–17
- Blocks, 5–21 to 5–22, 6–3 to 6–4
- Boolean algebra, 7–17
- Braces ({})
 - in compound statements, 5–21
 - in initializer lists, 8–18
- Bracket operator ([]), 7–5
- Breakpoint (DEBUG), 2–15
- Breakpoint (DEBUG MP)
 - on activation (multiprocess program), E–15
 - on termination (image exit), E–15
- break** statement, 6–6, 6–10
 - tutorial information (example), 4–12
- Built-in functions, 11–4 to 11–21
 - _ADAWI, 11–5
 - _BBCCI, 11–6
 - _BBSSI, 11–6
 - _FFC, 11–7
 - _FFS, 11–8
 - _HALT, 11–8
 - _INSQHI, 11–9
 - _INSQTI, 11–9
 - _INSQUE, 11–10
 - _LDPCTX, 11–10
 - _LOCC, 11–10
 - _MFPR, 11–11
 - _MOVC3, 11–11
 - _MOVC5, 11–12
 - _MOVPSL, 11–13
 - _MTPR, 11–14

Built-in functions (cont'd.)

- `_PROBER`, 11–14
- `_PROBEW`, 11–15
- `_READ_GPR`, 11–15
- `_REMQHI`, 11–16
- `_REMQTI`, 11–16
- `_REMQUE`, 11–17
- `_SCANC`, 11–17
- `_SIMPLE_READ`, 11–18
- `_SIMPLE_WRITE`, 11–19
- `_SKPC`, 11–19
- `_SPANC`, 11–20
- `_SVPCTX`, 11–21
- `_WRITE_GPR`, 11–21

builtins pragma, 10–23

Byte values, 8–8

BYTLM quota

- DEBUG MP requirements, E–26

C

C language

See also VAX C language
tutorial information, 4–1 to 4–2

`%c`

tutorial example using **printf**, 4–24

`C$INCLUDE` logical, 10–19

`C$LIBRARY` logical, 10–20

`CALL` command (DEBUG MP), E–9

calloc function

use in parallel processing, 3–29

Call stack, 2–14

`CANCEL MODULE` command (DEBUG), 2–35

`CANCEL SCOPE` command (DEBUG), 2–36

case label, 6–5

Case sensitivity, 5–17

tutorial information, 4–6

Cast operator, 7–13

`CC$float` predefined macro, 11–1

`CC$parallel` predefined macro, 11–2

`cc$rms_fab`

initialized RMS data structure, 12–8

`cc$rms_nam`

initialized RMS data structure, 12–8

`cc$rms_rab`

initialized RMS data structure, 12–8

`cc$rms_xaball`

initialized RMS data structure, 12–8

`cc$rms_xabdat`

initialized RMS data structure, 12–8

`cc$rms_xabfhc`

initialized RMS data structure, 12–8

`cc$rms_xabkey`

initialized RMS data structure, 12–8

`cc$rms_xabpro`

initialized RMS data structure, 12–8

`cc$rms_xabrdt`

initialized RMS data structure, 12–8

`cc$rms_xabsum`

initialized RMS data structure, 12–8

CC DCL command, 1–5

`/ANALYSIS_DATA` qualifier, 1–8

compilation errors, 1–20

`/CROSS_REFERENCE` qualifier, 1–8

`/DEBUG` qualifier, 1–9, 2–4

`/DEFINE` qualifier, 1–10 to 1–11, 1–18

`/DIAGNOSTICS` qualifier, 1–11

`/G_FLOAT` qualifier, 1–11

`/INCLUDE_DIRECTORY` qualifier, 1–12

`/LIBRARY` qualifier, 1–12

`/LIST` qualifier, 1–13

`/MACHINE_CODE` qualifier, 1–13

`/NOOPTIMIZE` qualifier, 2–4

`/OBJECT` qualifier, 1–14

`/OPTIMIZE` qualifier, 1–14

`//PARALLEL` qualifier, 1–15

`/PRECISION` qualifier, 1–15

`/PREPROCESS_ONLY` qualifier, 1–15

qualifiers for, 1–7 to 1–18

`/SHOW` qualifier, 1–16

`/STANDARD=PORTABLE` qualifier, 1–17

summary of, D–1

`/UNDEFINE` qualifier, 1–10 to 1–11, 1–18

`/WARNINGS` qualifier, 1–18

CDD

See Common Data Dictionary

cfree function

use in parallel processing, 3–29

Character

constants, 8–6

NUL character, 8–8

strings, 8–15, 8–20

See also, Arrays

debugger access to, 2–27

tutorial information, 4–21

variable declarations, 8–4

Character-string variables, 8–19

tutorial information, 4–21

char data type, 8–4

`CHAR_STRING_CONSTANTS` psect, 14–2 to 14–5

`$CLOSE` RMS function, 12–6

- \$CODE psect, 14–2 to 14–5
- Code replication, 3–20 to 3–21
 - See also, Data dependencies
- Command-line (DCL) arguments
 - conversion of, 5–17
 - to the VAX C main function, 5–15
- Command qualifiers
 - See CC DCL command
- Comma operator (,), 7–22
 - precedence of, 7–8
- Comments, 5–22
- Common Data Dictionary (CDD), 10–8 to 10–12
 - support for data types, 10–11
- Compilation process, 1–5 to 1–21
- Compilation unit
 - in determining scope, 9–2
- Compile DCL command
 - See CC DCL command
- Compiler messages, B–1 to B–53
- Compound statements, 5–21, 6–3
 - tutorial information, 4–11
- Conditional compilation, 10–13 to 10–15
- Conditional operator (?), 7–19
 - precedence of, 7–8
- Conditional statements, 6–4 to 6–7
- CONNECT command (DEBUG MP), E–5, E–17
- \$CONNECT RMS function, 12–6
- Constants, 8–1
 - character, 8–6
 - escape sequence, 8–7
 - hexadecimal escape sequence, 8–8
 - character strings, 8–20
 - floating-point, 8–10
 - identifiers in **#define** macros, 10–4
 - integer, 8–5
 - values of, 8–2
- const** modifier, 9–21
- continue** statement, 6–10
- Control dependency, 3–4
 - See also, Data dependencies
- Control flow statements, 6–1 to 6–2
- Conversions, 7–22 to 7–25
 - arithmetic, 7–22, 7–23
 - of data types, 7–22
 - of function arguments, 7–3, 7–24
 - summary of rules, D–7
 - with cast operator, 7–13
- \$CREATE RMS function, 12–6
- /CROSS_REFERENCE CC qualifier, 1–8

- C Run-Time Library (RTL)
 - See VAX C Run-Time Library (RTL)
- CTRL/C, 2–6
 - use in MPDEBUG, E–6
 - use in multiprocess debugger, E–12
- CTRL/Y, 2–6
 - use in multiprocess debugger, E–12, E–16

D

- D_floating representation, 8–9
- Data-class modifiers
 - description of, 9–21
- Data definitions and scope, 9–17
 - see also, Scope
- Data dependencies
 - algorithms to determine, 3–4
 - array variable references (example), 3–12
 - debugging example, E–21
 - definition of, 3–4
 - detecting existence of, 3–11 to 3–17
 - involving array variable references, 3–11
 - involving function calls, 3–13 to 3–14
 - involving pointer references, 3–15 to 3–16
 - involving scalar references, 3–16 to 3–17
 - Loop-carried, 3–4
 - Loop-independent, 3–4
 - pointer references (example), 3–15
 - recoding possible dependencies, 3–17 to 3–22
 - code replication, 3–20
 - loop alignment, 3–18
 - loop distribution, 3–21
 - related to program control flow, 3–4
 - scalar references (example), 3–16
- \$DATA psect, 14–2 to 14–5
- Data structures, 8–15
 - See also, Aggregates
 - RMS, 12–6
 - definition modules, 12–7
 - initialized prototypes, 12–7
- Data-type keywords
 - summary of, D–4
- Data-type modifiers
 - summary of, D–5
- Data types, 8–1 to 8–33
 - argument-conversion rules, 5–13
 - conversion of, 7–22
 - function prototypes, 5–9 to 5–11
 - modifiers, 9–21
 - tutorial information, 4–5
 - __DATE__ predefined macro, 11–3

- DBG\$PROCESS logical, E-2
 - defining, E-23
- DBG\$PROCESS logical name
 - program cycle for parallel processing, 3-7
- DBG> prompt, 2-5
- DCL commands
 - overview of program development, 1-1
 - summary of, D-1
- /DEBUG CC qualifier, 1-9, 2-4
- DEBUG command (DEBUG MP), E-16
- Debugger
 - access to program variables
 - arrays, 2-25
 - character strings, 2-27
 - scalars, 2-23
 - structures, 2-28
 - unions, 2-28
 - ASCII representation, 2-31
 - defined, 2-1
 - entering commands, 2-7
 - features of, 2-3
 - getting started with, 2-4
 - invoking, 2-5
 - invoking in multiprocess debugging session, E-2
 - keypad functions (figure), 2-8
 - multiprocess debugging example, E-21
 - prompt in multiprocessing session, E-3
 - sample session, 2-36 to 2-40
 - SHOW SYMBOL command, 2-29
 - system tuning for the multiprocess debugger, E-25
- Debugging configuration (DEBUG MP), E-2
- DEC/Shell, 10-16
 - inhibiting parallel processing, 3-10
- Declarations, 8-1 to 8-4
 - aggregate
 - arrays, 8-15
 - structures, 8-20
 - unions, 8-20
 - variant_struct**, 8-28
 - variant_union**, 8-28
 - determining scope, 9-2 to 9-4
 - format of, 8-3
 - function prototypes, 5-9 to 5-11
 - inside of blocks, 6-3
 - interpreting, 8-33 to 8-35
 - of VAX C functions, 5-7
 - parameters, 5-13
 - scalar
 - character constant, 8-6
 - character variable, 8-4

- Declarations
 - scalar (cont'd.)
 - enumerated, 8-13
 - integer, 8-4
 - pointer, 8-11
 - scope of, 5-9
 - vacuous tag declarations, 8-23
 - VAX C RTL prototypes, 5-11
 - void** functions, 8-32
- Declarators, 8-3
- Decomposition
 - conditions that inhibit, 3-9 to 3-10
 - definition of, 3-2
 - pragmas, 3-23 to 3-28
 - summary of (table), 3-5
 - syntax of, 10-22 to 10-28
- Decrement operator (--), 7-10
 - side effects within macros, 10-6
 - tutorial information, 4-16
- default** label, 6-5
- DEFINE/PROCESS_GROUP command (DEBUG MP), E-14
- /DEFINE CC qualifier, 1-10
 - examples and usage, 1-18
- #define** directive, 10-2, 10-4
- defined** operator, 10-15
- Definition modules
 - descriptions of, A-1 to A-8
 - for RMS structures, 12-7
 - table of, A-1
- Definitions
 - See also, Declarations
 - affect on scope, 9-2
 - of constants, 8-1
 - of functions, 5-1
 - of **void** functions, 5-5, 8-32
- \$DELETE RMS function, 12-6
- Dependencies
 - See Data dependencies
- DEPOSIT command (DEBUG), 2-21
- Dereferencing pointers, 7-11
 - See also Pointers
- descrip* text-library module, 13-14
- \$DESCRIPTOR preprocessor macro, 13-18
- Descriptors
 - defined, 13-7
 - in mixed-language programming, 13-14
- /DIAGNOSTICS CC qualifier, 1-11
- #dictionary** directive, 10-8, 10-9, 10-12
- Direct access modes (RMS), 12-4

Directives

- #define**, 10–2
- #dictionary**, 10–8
- #elif**, 10–13
- #else**, 10–13
- #endif**, 10–13
- #if**, 10–13
- #ifdef**, 10–13
- #ifndef**, 10–13
- #include**, 10–16
- #line**, 10–21
- #module**, 10–21
- #pragma**, 10–22
- #undef**, 10–4

\$DISCONNECT RMS function, 12–6

Display (DEBUG)
source code, 2–10

Display (DEBUG MP)
process specific, E–17

Division operator (/), 7–15

DO command (DEBUG MP), E–6, E–8

do statement, 6–9
tutorial information, 4–14

double data type, 8–9

Dynamic module setting in the debugger, 2–35

Dynamic process setting (DEBUG MP), E–10

Dynamic prompt setting (DEBUG MP), E–3

E

ECHO DCL command, 5–16

Editing
LSE, C–1 to C–26
TPU, 1–4
EVE interface, 1–4

#elif preprocessor directive, 10–13, 10–14

Ellipses, 5–6

#else preprocessor directive, 10–13

#endif preprocessor directive, 10–13

ENQLM quota
DEBUG MP requirements, E–25

Enumerated data type, 8–13 to 8–15
declaration of, 8–13

enum type, 8–13

envp main function argument, 5–15

Equality operators, 7–16

Equal-to operator (=), 7–16

\$ERASE RMS function, 12–6

errno
checking during parallel processing, 3–10
parallel **malloc** behavior, 3–30

Errors

See also, CC DCL command

See also, LINK DCL command

compiler messages and descriptions, B–1 to B–53

during compilation, 1–20 to 1–21

link-time, 1–30

run-time, 1–32

Escape sequences, 8–7
hexadecimal values, 8–8
summary of, D–8

EVALUATE command (DEBUG), 2–21

Evaluating expressions
See Expressions

Evaluation order
in argument lists, 5–12

EVE
See Editing

EXAMINE command (DEBUG), 2–20

Exception handling
inhibiting parallel processing, 3–10

Execution
start/resume in debugger, 2–12
start/resume in DEBUG MP, E–7
suspending with watchpoint (DEBUG MP), E–19

EXIT command (DEBUG MP), E–11

exponentiation operator, Glossary–8

Expressions, 6–3 to 6–4, 7–1 to 7–25
See also, Address expression
See also, Language expression

assignment, 7–20

as statements, 6–3

binary

- additive, 7–15
- bitwise, 7–17
- equality, 7–16
- logical, 7–17
- multiplication, 7–15
- relational, 7–16
- shift, 7–19

comma, 7–22

conditional, 7–19

evaluation order

- ambiguity of, 7–11

primary, 7–3 to 7–6

- array reference, 7–5
- function call, 7–3
- lvalues, 7–2
- parentheses, 7–3
- rvalues, 7–2

- Expressions
 - primary (cont'd.)
 - structure reference, 7–6
 - syntax of, 7–3
 - union reference, 7–6
 - unary
 - addressed, 7–11
 - cast, 7–13
 - increment and decrement, 7–10
 - negation, 7–10
 - one's complement, 7–12
 - sizeof**, 7–14
- Extended attribute block–XAB (RMS)
 - initialization of, 12–11
- Extensible VAX Editor (EVE)
 - See Editing
- External storage class
 - compared to global, 9–17 to 9–19
 - data definitions, 9–17
 - [extern]**, 9–13
- [extern]** keyword, 9–13 to 9–14
 - affect on parallel processing, 3–13, 3–22
 - aligning, 3–22
 - overlapping psects, 9–18
 - table of psect attributes, 14–3
 - scope of, 9–5

F

- \f (form feed), 8–7
- F_floating representation, 8–9
- fab definition module, 12–7
- FAB RMS data structure, 12–6
 - initialization of, 12–9
- _FFC built-in function, 11–7
- _FFS built-in function, 11–8
- __FILE__ predefined macro, 11–3
- Files (RMS)
 - indexed organization, 12–3
 - organization, 12–2 to 12–4
 - relative organization, 12–3
 - sequential organization, 12–2
- FILLM quota
 - DEBUG MP requirements, E–26
- float data type, 8–9
- Floating-point
 - constants, 8–10
 - data type
 - declaration of, 8–9
 - double**, 8–9
 - D_floating, 8–9

- Floating-point
 - data type (cont'd.)
 - F_floating, 8–9
 - G_floating, 8–9
 - long**, 8–9
 - precision of, 8–9
 - passed by immediate value, 13–10
 - sizes and ranges of, 8–9
- fopen** function
 - example of, 5–2
- FOR\$PROCESSES logical name
 - See also, Parallel processing
 - preparing programs for parallel processing, 3–7
 - use of, 3–31
- FOR\$SPIN_WAIT logical name
 - See also, Parallel processing
 - use of, 3–32
- FOR\$STALL_WAIT logical name
 - See also, Parallel processing
 - use of, 3–33
- Foreign command, 12–14
 - for passing command-line arguments, 5–15
- for** keyword
 - loop decomposition in parallel processing, 3–2
- for** statement, 6–8
 - tutorial example of, 4–15
 - tutorial information, 4–15
- FORTRAN common block
 - sharing program sections with, 13–37
- Forward referencing
 - function declarations (example), 5–7
 - structures, 8–23
- free** function
 - use in parallel processing, 3–29
- frexp** function
 - inhibiting parallel processing, 3–14
- Function
 - address of, 5–14, 7–4
 - argument-conversion rules, 5–13
 - as arguments, 5–14
 - built-in, 11–4 to 11–21
 - calls between programs of different languages, 13–19
 - calls to, 7–3
 - within macros, 10–6
 - data dependencies introduced during parallel processing, 3–13
 - declaring VAX C functions, 5–7
 - definitions of, 5–1 to 5–13
 - argument conversion, 7–3
 - arguments, 5–2, 5–12

Function

definitions of (cont'd.)

- body, 5–2
- main function, 5–3
- main_program option, 5–3
- names of, 5–3
- parameters, 5–2, 5–12

identifiers, 7–4

implicit declaration of, 5–3

parallel-processing memory-management restrictions, 3–29

parameter-passing mechanisms, 13–6

prototypes, 5–9 to 5–11

return values of, 5–7

RMS, 12–5

scope of, 5–3

tutorial information, 4–4

undeclared, 7–3

VAX C RTL prototypes, 5–11

void, 8–32

Function arguments

conversion of, 7–24, D–7

Function prototypes

See also, Function

scope rules, 5–11

widening rules, 5–11

G

/G_FLOAT CC qualifier, 1–11

G_floating representation, 8–9

GBLPAGES parameter

DEBUG MP requirements, E–26

GBLPAGES parameter (VMS)

tuning for parallel processing, 3–35

GBLPAGFIL parameter (VMS)

tuning for parallel processing, 3–35

GBLSECTIONS parameter (VMS)

DEBUG MP requirements, E–26

tuning for parallel processing, 3–34

getchar

tutorial information (example), 4–10

\$GET RMS function, 12–6

globaldef keyword

used in declarations inside of blocks, 6–3

globaldef specifier, 9–15, 9–17

loading modules with global definitions, 9–18

with enumerated values, 9–20

Global pages

use by DEBUG MP, E–26

globalref specifier, 9–15, 9–17

globalref specifier (cont'd.)

See also, Storage classes

loading modules with global definitions, 9–18

with enumerated values, 9–20

Global sections

use by DEBUG MP, E–26

Global section watchpoint (DEBUG MP), E–19

Global storage class, 9–15 to 9–20

compared to **extern**, 9–17 to 9–19

variable initialization, 9–15

Global symbol (VMS)

to test status values, 13–59

globalvalue specifier, 9–19

GO command (DEBUG), 2–12

GO command (DEBUG MP), E–7

goto statement, 6–2

Greater-than operator (>), 7–16

Greater-than or equal-to operator(>=), 7–16

H

_HALT built-in function, 11–8

HELP (DEBUG)

online, 2–3

Hexadecimal

byte values, 8–8

/HOLD qualifier (DEBUG MP)

SET PROCESS command, E–4, E–8

I

Identifiers, 5–17 to 5–18

#ifdef preprocessor directive, 10–13

#ifndef preprocessor directive, 10–13

#if preprocessor directive, 10–13

using the **defined** operator, 10–15

if statement, 6–4

tutorial information, 4–10

ignore_dependency decomposition pragma

syntax of, 10–23

use of, 3–25

ignore_dependency pragma

use of, 3–16

#include preprocessor directive, 10–16, 10–20

default text libraries, 1–6

descrip module, 13–14

RMS data-structure inclusion, 12–7

/INCLUDE_DIRECTORY CC qualifier, 1–12

Including files, 10–16 to 10–20

VAX C RTL prototypes, 5–11

Increment operator (++), 7–10

- Increment operator (++) (cont'd.)
 - side effects within macros, 10–6
 - tutorial information, 4–16
- Indexed file organization (RMS), 12–3
- Indirection operator (*), 7–11, 8–11
 - tutorial information, 4–18
- Initialization
 - arrays, 8–16
 - characters, 8–4
 - character-string variables, 8–19
 - debugger, 2–5
 - integers, 8–4
 - of global variables, 9–15
 - of RMS data structures
 - extended attribute block (XAB), 12–11
 - file access block (FAB), 12–9
 - name block (NAM), 12–12
 - record access block (RAB), 12–10
 - structures, 8–26
 - unions, 8–26
- Initialized RMS data structure
 - cc\$rms_fab, 12–8
 - cc\$rms_nam, 12–8
 - cc\$rms_rab, 12–8
 - cc\$rms_xaball, 12–8
 - cc\$rms_xabdat, 12–8
 - cc\$rms_xabfbc, 12–8
 - cc\$rms_xabkey, 12–8
 - cc\$rms_xabpro, 12–8
 - cc\$rms_xabrdt, 12–8
 - cc\$rms_xabsum, 12–8
- inline** pragma, 10–24
- Input/output (I/O)
 - during parallel processing, 3–10
 - RMS, 12–1
 - tutorial information, 4–6
- _INSQHI built-in function, 11–9
- _INSQTI built-in function, 11–9
- _INSQUE built-in function, 11–10
- int** data type, 8–4
 - tutorial information, 4–5
- Integer constants, 8–5
 - invalid, 8–6
- Integer data types, 8–4
- Internal storage class, 9–9 to 9–12
- Interrupt (DEBUG)
 - execution of command, 2–6
 - execution of program, 2–6
- Interrupting statements, 6–10 to 6–11
- ISAM (RMS)
 - indexed-sequential access method, 12–5

K

- Keypad key definitions
 - multiprocess debugger predefined, E–18
- Keywords
 - auto**, 9–10
 - break** statement, 6–10
 - case** statement, 6–5
 - char**, 8–4
 - const**, 9–21
 - continue** statement, 6–10
 - default** label, 6–5
 - description of (table), 5–18
 - description of data types (table), 8–3
 - do** statement, 6–9
 - double**, 8–9
 - else** statement, 6–4
 - enum**, 8–13
 - extern**, 9–13
 - float**, 8–9
 - for** statement, 6–8
 - globaldef**, 9–15
 - globalref**, 9–15
 - globalvalue**, 9–19
 - goto** statement, 6–2
 - if** statement, 6–4
 - int**, 8–4
 - noshare**, 9–24
 - readonly**, 9–25
 - register**, 9–11
 - return** statement, 6–11
 - short**, 8–4
 - sizeof**, 7–14
 - static**, 9–12
 - struct**, 8–20
 - switch** statement, 6–5
 - tutorial information, 4–6
 - typedef**, 8–32
 - union**, 8–20
 - variant_struct**, 8–28
 - variant_union**, 8–28
 - void**, 8–13, 8–32
 - volatile**, 9–23
 - while** statement, 6–9

L

- Label statements, 6–2
- Language expression
 - with DEPOSIT command (DEBUG), 2–21
 - with EVALUATE command (DEBUG), 2–21

- Language-Sensitive Editor (LSE)
 - See Editing
- `_LDPC_TX` built-in function, 11–10
- Less-than operator (<), 7–16
- Less-than or equal-to operator (<=), 7–16
- Lexical scope, 9–4 to 9–6
- Libraries
 - default object-module file types, 1–26
 - default text-module file types, 1–6
 - inclusion of text modules, 10–19
 - VAX C RTL object-module linking order, 1–27
- `/LIBRARY CC` qualifier, 1–12
- Lifetime of stored objects, 9–8
- Limit of nested **#include** lines, 10–16
- `__LINE__` predefined macro, 11–3
- Line number
 - SET BREAK command (DEBUG), 2–15
 - SET TRACE command (DEBUG), 2–17
 - source display (DEBUG), 2–11
- #line** preprocessor directives, 10–21
- `/LINE` qualifier
 - to the debugger SET TRACE command, 2–17
- LINK command
 - tutorial information (example), 4–9
- LINK DCL command, 1–22
 - `/DEBUG` qualifier, 2–4
 - link-time errors, 1–30 to 1–31
 - qualifiers, 1–23
 - summary of, D–3
 - VAX C RTL object-module linking order, 1–27
 - VAX C RTL shareable images, 1–29
- Link-time scope, 9–4 to 9–6
- LINT, 5–22
 - function prototypes, 5–9
- `/LIST CC` qualifier, 1–13
- LNK\$LIBRARY logical, 1–27
- `_LOCC` built-in function, 11–10
- Logical arithmetic
 - negation operator, 7–10
 - operators, 7–17
- Logical OR operator
 - tutorial information, 4–21
- long** data type, 8–4, 8–9
- longjmp** function
 - inhibiting loop decomposition, 3–10
- Loop alignment, 3–18
 - See also, Data dependencies
- Loop-carried dependencies, 3–4
 - See also, Data dependencies

- Loop decomposition
 - See Decomposition
- Loop distribution, 3–21 to 3–22
 - See also, Data dependencies
- Loop-independent dependencies, 3–4
 - See also, Data dependencies
- Looping statements, 6–7 to 6–10
 - See also, Statements
 - code replication, 3–20 to 3–21
 - data dependencies (definition of), 3–11
 - decomposition (definition), 3–2
 - detecting data dependencies, 3–17
 - loop distribution, 3–21 to 3–22
 - realigning possible data dependencies, 3–18
 - tutorial information, 4–14
- LSE
 - See Editing
- lvalues, 7–2
 - tutorial information, 4–17

M

- `/MACHINE_CODE CC` qualifier, 1–13
- MACRO program
 - sharing program sections with, 13–41
- Macros
 - definitions, 10–2
 - canceling, 10–4
 - listing substituted lines, 10–8
 - naming parameters in, 10–6
 - parameters within definitions, 10–4
 - possible side effects, 10–6
 - predefined
 - `__align` boundaries, 9–25
 - `CC$gfloat`, 11–1
 - `CC$parallel`, 11–2
 - `__DATE__`, 11–3
 - `__FILE__`, 11–3
 - `__LINE__`, 11–3
 - `__TIME__`, 11–3
 - vax, 11–4
 - vax11c, 11–4
 - vaxc, 11–4
 - vms, 11–4
 - substitution within **#include** directives, 10–20
 - tutorial information, 4–7
- Main function, 5–3 to 5–4
 - main_program option, 5–3
 - passing arguments to, 5–15
 - syntax of, 5–15
- main_program option, 5–3

malloc function
 use in parallel processing, 3–29

math.h function calls
 parallel processing support, 3–14

Members
 defined, 8–2
 variant aggregates, 8–28

member_alignment pragma, 10–25

Memory-management functions
 parallel-processing versions, 3–29

Messages
 See also, Errors
 compiler, B–1 to B–53
 format of (compiler), 1–20 to 1–21

_MFPR built-in function, 11–11

Mixed-language programming, 5–12
 argument passing
 by descriptor, 13–14
 by immediate value, 13–8
 floating-point numbers, 13–10
 by reference, 13–11
 return status values, 13–54
 format, 13–54
 manipulating, 13–56
 system service, 13–50
 testing, 13–58
 variable-length argument lists, 13–52

VAX Calling Standard, 13–2
 affect on VAX C functions, 5–12

modf function
 inhibiting parallel processing, 3–14

Modifiers
 storage class, 9–23

#module preprocessor directive, 10–21

Modules
 changing the default name, 10–21
 default object-library file types, 1–26
 VAX C RTL object linking order, 1–27

Module setting (DEBUG), 2–35

Modulo operator (%), 7–15

_MOVC3 built-in function, 11–11

_MOVC5 built-in function, 11–12

_MOVPSL built-in function, 11–13

_MTPR built-in function, 11–14

Multiplication operators (*), 7–15

N

\n (newline), 8–7

NAM RMS data structure, 12–6
 initialization of, 12–12

Negation
 arithmetic and logical, 7–10

Nesting of #include lines, 10–16

%NEXT_PROCESS logical (DEBUG MP), E–13

noinline pragma, 10–24

nomember_alignment pragma, 10–25

/NOOPTIMIZE qualifier
 effect on debugging, 2–4

Noscreen mode (DEBUG), 2–10

noshare modifier, 9–24

nostandard pragma, 10–28

Not-equal-to operator (!=), 7–16
 tutorial information (example), 4–15

NUL character, 8–8

Null pointer, 8–11
 used with the equality operator, 7–21

NULL predefined macro, 8–11

Null statement, 6–1

NUL-terminated strings, 8–20

O

/OBJECT CC qualifier, 1–14

Object module
 default library file types, 1–26
 in determining scope, 9–2
 names provided after run-time errors, 1–32
 VAX C RTL linking order, 1–27

Objects of variables, 8–2

Octal constants, 8–5

One's complement operator (~), 7–12

\$OPEN RMS function, 12–6

Operand conversion, 7–23

Operators, 7–6 to 7–22
 assignment, 7–20 to 7–22
 ambiguity of, 7–21
 binary, 7–14 to 7–19
 additive, 7–15
 bitwise, 7–17
 equality, 7–16
 logical, 7–17
 multiplication, 7–15
 relational, 7–16
 shift, 7–19
 bracket, 7–5
 categories of, 7–8
 comma, 7–22
 conditional, 7–19
 defined, 10–15
 exponentiation, Glossary–8
 indirection, 8–11

Operators (cont'd.)

- list of, 7-6
- precedence of, 7-8, D-5
- unary, 7-8, 7-10 to 7-14
 - address of, 7-11
 - cast, 7-13
 - increment and decrement, 7-10
 - indirection, 7-11
 - negation, 7-10
 - one's complement, 7-12
- /OPTIMIZE CC qualifier, 1-14
- /OPTIMIZE qualifier
 - to the CC DCL command, 2-38
- OR bitwise operator (|), 7-17

P

Parallel processing, 3-1 to 3-37

- See also, Decomposition
- across time (figure), 3-2
- checking *errno* values, 3-10
- data dependency analysis, 3-6
- debugging example, E-21
- decomposition pragmas, 3-23 to 3-28
 - summary of (table), 3-5
 - syntax of, 10-22 to 10-28
- description of subprocess creation, 3-3
- FOR\$PROCESSES logical, 3-31
- FOR\$SPIN_WAIT logical, 3-32
- FOR\$STALL_WAIT logical, 3-33
- inhibiting behavior, 3-9 to 3-10
 - DEC/Shell, 3-10
 - of exception handling, 3-10
 - of I/O operations, 3-10
 - of signal handling, 3-10
- linking against VAXCPAR.OLB library, 1-27, 3-29
- memory-management functions, 3-29 to 3-30
- overview, 3-1 to 3-6
- preparing programs for (compiling), 3-6 to 3-8
 - summary (figure), 3-7
- rewriting possible dependencies, 3-17
 - code replication, 3-20 to 3-21
 - loop alignment, 3-18 to 3-20
 - loop distribution, 3-21 to 3-22
- run-time logical names, 3-30 to 3-33
 - FOR\$PROCESSES, 3-31
 - FOR\$SPIN_WAIT, 3-32
 - FOR\$STALL_WAIT, 3-33
 - summary of (table), 3-6
- setting up the environment, 3-3
- storage classes effect on, 3-22

Parallel processing (cont'd.)

- suitable applications for, 3-2
- support for *math.h* functions, 3-14
- tuning, 3-30 to 3-37
 - with the AUTHORIZE utility, 3-36
 - with the GBLPAGES parameter, 3-35
 - with the GBLPAGFIL parameter, 3-35
 - with the GBLSECTIONS parameter, 3-34
 - with the SYSGEN Utility, 3-33
 - with the working set size, 3-37
- VAX C support mechanisms (table), 3-5
- /PARALLEL qualifier
 - preparing programs for parallel processing, 3-6
- Parameters, 5-12 to 5-13
 - declaration of, 5-2, 5-13
 - function prototypes, 5-9
 - in #define preprocessor macros, 10-4
 - main function, 5-15
 - passing
 - by descriptor, 13-7
 - by immediate value, 13-7
 - by reference, 13-7
 - passing mechanisms in mixed-language programming, 13-6
 - rules governing, 5-12
 - tutorial information, 4-5
- Parenthetical expressions, 7-3
- Path name
 - in debugging, 2-10, 2-13, 2-15, 2-36
- PC
 - and source display, 2-11
- PC (DEBUG)
 - and SHOW CALLS display, 2-14
 - and STEP command, 2-13
 - breakpoint, 2-15
- Period operator (.), 7-6
- PGFLQUO quota
 - DEBUG MP requirements, E-25
 - tuning for parallel processing, 3-36
- PL/I externals
 - sharing program sections with, 13-39
- Pointers
 - arithmetic, 8-11
 - data dependencies introduced during parallel processing, 3-15
 - declaration of, 8-11
 - legal operations, 8-11
 - null, 8-11
 - tutorial information, 4-17
 - unary operator, 7-11
 - using the increment operator (++), 7-10

Pointers (cont'd.)

void, 8–13

Portability concerns

See also, C language

accessing argument lists, 13–7

char * generic-pointer notation, 8–13

character-string constants, 8–7

character-string length, 8–20

comparing pointers and integers, 7–16

conversion of command-line arguments, 5–17
defined, 4–3

deviations of assignment operators, 7–21

#dictionary directive, 10–9

direction of bit field packing, 8–30

global storage classes, 9–17

global system status values, 9–19

#include using angle brackets, 10–18

int values on a VAX, 8–5

length of argument list, 5–12

length of bit fields, 8–30

length of identifiers, 5–17

lexical scope and compilation units, 9–2

long float keywords, 8–9

main_program option, 5–3

#module directive, 10–21

modules with **extern** definitions, 9–18

nested **#include** files, 10–18

octal constants, 8–5

parameter declarations, 5–13

passing constants by reference, 13–12

predefined symbols, 5–20

predefined system-definition macros, 11–1

preprocessor implementations, 10–1

referencing aggregate members, 8–25

structure-member alignment, 8–21

text modules in the **#include** line, 10–19

tutorial information, 4–2 to 4–4

UNIX file specifications, 10–16

variant structures and unions, 8–30

#pragma preprocessor directive, 10–22

Pragmas

builtins, 10–23, 11–5

decomposition, 3–23 to 3–28

ignore_dependency syntax, 10–23

placement rules, 3–23

safe_call syntax, 10–26

sequential_loop syntax, 10–27

summary of pragmas (table), 3–5

using **ignore_dependency**, 3–25

using **safe_call**, 3–14, 3–26, 3–28

inline, 10–24

Pragmas (cont'd.)

member_alignment, 10–25

standard, 10–28

PRCLM quota

DEBUG MP requirements, E–25

Precedence of operators, 7–8

in interpreting declarations, 8–33

/PRECISION CC qualifier, 1–15

Predefined macros, 11–1 to 11–4

NULL macro, 8–11

Predefined symbols, 5–20

Preprocessor directives, 10–1 to 10–28

#define, 10–2

#dictionary, 10–8

#elif, 10–13

#else, 10–13

#endif, 10–13

#if, 10–13

#ifdef, 10–13

#ifndef, 10–13

#include, 10–16
macro substitution, 10–20

#line, 10–21

#module, 10–21

#pragma, 10–22

summary of, D–8

#undef, 10–4

/PREPROCESS_ONLY CC qualifier, 1–15

%PREVIOUS_PROCESS logical (DEBUG MP), E–13

Primary expressions, 7–3 to 7–6

See also, Expressions

array reference, 7–5

function call, 7–3

lvalues, 7–2

parentheses, 7–3

structure reference, 7–6

union reference, 7–6

Primary operators

precedence of, 7–8

printf function

tutorial information (example), 4–7

tutorial information on using %c (example), 4–24

tutorial information on using %s (example), 4–21

Privacy of data

See Scope

_PROBER built-in function, 11–14

_PROBEW built-in function, 11–15

Processes (DEBUG MP)

activation tracepoint, predefined, E–15

connecting debugger to, E–17

connecting to, E–5

Processes (DEBUG MP) (cont'd.)
 multiprocess debugging, E-1
 termination tracepoint, predefined, E-15
/PROCESS qualifier (DEBUG MP)
 DO command, E-6
 SET DISPLAY and DISPLAY commands, E-18
/PROCESS_GROUP qualifier (DEBUG MP)
 DEFINE command, E-14
%PROCESS_NAME logical (DEBUG MP), E-13
%PROCESS_NUMBER logical (DEBUG MP), E-13
%PROCESS_PID logical (DEBUG MP), E-13
Program counter (PC)
 See PC
Program section (psect)
 attributes of, 14-1 to 14-5
 attributes of (table), 14-3
 comparing global and external classes, 9-17
 created by VAX C, 14-2
 for global symbols, 9-15
 sharing
 with FORTRAN common blocks, 13-37
 with MACRO programs, 13-41
 with PL/I externals, 13-39
Program structure, 5-1 to 5-23
Promotion of data types, 7-22
Prototypes, 5-9
 for VAX C RTL functions, 5-11
Psect
 See Program Section
PSL, 11-14
\$PUT RMS function, 12-6

Q

Qualifiers

 CC command, 1-7
 LINK command, 1-23
 position of, 1-5
 summary of CC command, D-2
 summary of LINK command, D-3
QUIT command (DEBUG MP), E-11
Quotas
 See User quotas

R

\r (carriage return), 8-7
RAB RMS data structure, 12-6
 initialization of, 12-10
Random access mode (RMS), 12-4
_READ_GPR built-in function, 11-15

readonly modifier, 9-25
 realloc function
 use in parallel processing, 3-29
Record file address (RMS)
 access mode, 12-4
Record Management Services (RMS), 12-1 to 12-31
 data structures, 12-6
 example program, 12-13
 extended attribute blocks, 12-6
 file access blocks, 12-6
 file organization, 12-2 to 12-4
 functions, 12-5
 summary of, D-9
 indexed organization, 12-3
 name blocks, 12-6
 random access mode, 12-4
 record access blocks, 12-6
 record access modes, 12-4
 record formats, 12-5
 relative organization, 12-3
 return status values, 12-7
 sequential organization, 12-2
register keyword
 restrictions, 8-12
 used in declarations inside of blocks, 6-3
 used with the address of operator (&), 7-12
register storage class, 9-11
Relational operators, 7-16
Relative RMS file organization, 12-3
_REMQHI built-in function, 11-16
_REMQTI built-in function, 11-16
_REMQUE built-in function, 11-17
Reserved words, 5-18
return keyword
 example of, 5-2
 in function definitions, 5-2
 statement syntax, 6-11
return statement
 tutorial information, 4-5
Return status (VMS)
 severity codes, 13-55
 value
 format of, 13-54
 manipulating, 13-56
 RMS, 12-7
 system service, 13-50
 testing for specific value, 13-59
 testing for success or failure, 13-58
\$REWIND RMS function, 12-6
Right arrow operator (->), 7-6

RMS
 See Record Management Services (RMS)
 rmsdef definition module, 12–7
 rms definition module, 12–7
 RST (run-time symbol table)
 as used by the debugger, 2–34
 RTL
 See VAX C Run-Time Library (RTL)
 See VMS Run-Time Library (RTL)
 RUN DCL command, 1–31
 invoking and terminating the debugger, 2–5
 run-time errors, 1–32
 Run-time errors, 1–32
 See also, Errors
 Run-Time Library (RTL)
 See VAX C Run-Time Library (RTL)
 See VMS Run-Time Library (RTL)
 rvalues, 7–2
 tutorial information, 4–17

S

safe_call pragma
 syntax of, 10–26
 used in *math.h*, 3–14
 use of, 3–14, 3–26, 3–28
 SCA, C–20 to C–26
 Scalar data types, 8–4 to 8–15
 data dependencies introduced during parallel
 processing, 3–16
 debugger access to, 2–23
 declarations, 8–4
 character, 8–4
 enumerated, 8–13
 floating-point, 8–9
 integer, 8–4
 pointers, 8–11
 defined, 8–2
 tutorial information, 4–21
 _SCANC built-in function, 11–17
 Scope, 9–1 to 9–8
 auto variables, 5–21
 in a compilation unit, 9–2
 in an object module, 9–2
 in a program, 9–2
 lexical scope, 9–4
 link-time scope, 9–4
 of external data, 9–10
 of functions, 5–3
 position of declarations, 9–2 to 9–4

Screen mode
 debugger, 2–11
 multiprocess debugger, E–17
 Sequential access mode (RMS), 12–4
 Sequential file organization (RMS), 12–2
 Sequential program execution, 3–2
 See also, Parallel processing
sequential_loop pragma
 syntax of, 10–27
 SET ABORT_KEY command, 2–7
 SET ABORT_KEY command (DEBUG MP), E–12
 SET BREAK command (DEBUG), 2–15
 SET MODE [NO]DYNAMIC command (DEBUG),
 2–35
 SET MODE [NO]INTERRUPT command (DEBUG MP),
 E–7
 SET MODULE command (DEBUG), 2–10, 2–35
 SET PROCESS command (DEBUG MP), E–8, E–9
 SET SCOPE command (DEBUG), 2–36
 SET SOURCE command (DEBUG), 2–11
 SET TRACE command (DEBUG), 2–17
 SET WATCH command (DEBUG), 2–18
 Shareable images
 linking against the VAX C RTL, 1–29
 /SHARE LINK qualifier, 1–29
 /SHARE qualifier
 to the debugger SET TRACE command, 2–17
 Shift operators, 7–19
 Shift operators (<< and >>), 7–19
short data type, 8–4
 SHOW CALLS command (DEBUG), 2–14
 /SHOW CC qualifier, 1–16
 SHOW MODULE command (DEBUG), 2–35
 SHOW SCOPE command (DEBUG), 2–36
 SHOW SYMBOL command (DEBUG), 2–29, 2–36
 Signal handling
 inhibiting parallel processing, 3–10
 /SILENT qualifier
 to the debugger SET TRACE command, 2–17
 _SIMPLE_READ built-in function, 11–18
 _SIMPLE_WRITE built-in function, 11–19
sizeof keyword, 7–14
 _SKPC built-in function, 11–19
 Source Code Analyzer
 See SCA
 Source code display (DEBUG MP), E–17
 Source display (DEBUG), 2–10, 2–11
 TYPE command (DEBUG), 2–10
 Source lines (DEBUG)
 not available, 2–10, 2–12
 _SPANC built-in function, 11–20

SS\$_NORMAL

condition value, 13–50

/STANDARD=PORTABLE CC qualifier, 1–17

standard pragma, 10–28

Statements, 6–1 to 6–11

break, 6–5, 6–10

case, 6–5

compound, 6–3

conditional, 6–4 to 6–7

continue, 6–10

control flow, 6–1 to 6–2

default, 6–5

do, 6–7, 6–9

expressions as, 6–3

for, 6–7, 6–8

goto, 6–2

if, 6–4

interrupting, 6–10 to 6–11

labels, 6–2

like, 6–7

looping, 6–7 to 6–10

null, 6–1

return, 6–11

summary of, D–6

switch, 6–5

while, 6–9

static keyword

used in declarations inside of blocks, 6–3

static storage class, 9–12

stddef module

parallel-processing version, 3–29

STEP command (DEBUG), 2–13

STEP command (DEBUG MP), E–7

Storage allocation, 9–8 to 9–9

for program sections, 9–8

attributes of, 14–1

lifetime of variables, 9–8

location of, 9–9

registers, 9–8

run-time stack, 9–8

Storage classes, 9–1 to 9–26

affect on parallel processing, 3–22

defined, 9–1

external, 9–13 to 9–14

definitions and declarations, 9–13

global, 9–15

in determining scope, 9–1

internal, 9–9

[auto], 9–10

register, 9–11

list of classes and specifiers (table), 9–3

Storage classes (cont'd.)

list of specifiers and their scope (table), 9–5

modifiers, 9–23 to 9–26

concept defined, 9–4

const, 9–21

noshare, 9–24

readonly, 9–25

volatile, 9–23

order of keywords in declarations, 9–3

psect attributes (table), 14–3

specifiers

auto, 9–10

[auto], 9–5

[extern], 9–5

globaldef, 9–7

globalref, 9–15

globalvalue, 9–19

(null), 9–5

register, 9–11

static, 9–7

static, 9–12

tutorial information, 4–5

Storage-class modifiers

description of, 9–23

summary of, D–5

Storage-class specifiers

description of, 9–1

summary of, D–5

strcpy function

copying character strings (example), 8–19

tutorial information (example), 4–23, 4–24

String data type

See also, Arrays

copying, 8–19

declaration of, 8–19

NUL-terminated strings, 8–20

strncpy function

copying character strings (example), 8–19

struct keyword, 8–20

Structures, 8–15

bit fields, 8–30

debugger access to, 2–28

declaration of, 8–20, 8–22 to 8–24

forward referencing, 8–23

initialization

tutorial example of, 4–23

initialization of, 8–26

legal and illegal operations on, 8–21

members of

references to, 7–6, 8–24 to 8–25

passed by descriptor, 13–14

- Structures (cont'd.)
 - tutorial information, 4–22
 - variant aggregates, 8–28
- Substitution
 - macro, 10–4, 10–5
 - within **#include** directives, 10–20
- Subtraction operator (–), 7–10
- /SUFFIX qualifier (DEBUG MP), E–18
- _SVPCTX built-in function, 11–21
- switch** keyword
 - declarations inside of, 6–7
- switch** statement, 6–5, 6–7
 - tutorial example of, 4–12
 - tutorial information, 4–12
- Symbol
 - module setting, 2–35
 - record, 2–4
 - relation to path name, 2–13
- Symbolic constants, 8–1
- sys\$close RMS function, 12–6
- sys\$connect RMS function, 12–6
- sys\$create RMS function, 12–6
- sys\$delete RMS function, 12–6
- sys\$disconnect RMS function, 12–6
- sys\$erase RMS function, 12–6
- sys\$get RMS function, 12–6
- SYSLIBRARY logical, 10–19
- sys\$open RMS function, 12–6
- sys\$put RMS function, 12–6
- sys\$rewind RMS function, 12–6
- sys\$update RMS function, 12–6
- SYSGEN Utility
 - tuning for parallel processing, 3–33 to 3–35
- System parameters
 - tuning for parallel processing, 3–33 to 3–35
- /SYSTEM qualifier
 - to the debugger SET TRACE command, 2–17
- System resources
 - DEBUG MP requirements, E–25 to E–26
 - VAX C requirements, 3–33

T

- \t (horizontal tab), 8–7
- Tags of structures and unions, 8–22
 - See also, Structures
 - See also, Unions
 - vacuous declarations, 8–23
- /TERMINATING qualifier (DEBUG MP)
 - SET TRACE command, E–15
- Termination (DEBUG MP), E–11, E–15

- Text libraries, 1–6
 - default module file types, 1–6
 - VAXCDEF.TLB
 - tutorial information, 4–7
 - __TIME__ predefined macro, 11–3
 - Token replacement, 10–2
 - _tolower** macro
 - tutorial information (example), 4–12
 - _toupper** macro, 10–4, 10–6
 - TPU
 - See Editing
 - Traceback
 - See also, Debugger
 - See also, Errors
 - run-time errors, 1–32
 - Traceback (DEBUG)
 - SHOW CALLS command (DEBUG), 2–14
 - Tracepoint (DEBUG), 2–17
 - Tracepoint (DEBUG MP)
 - on activation (multiprocess program), E–15
 - on termination (image exit), E–15
 - predefined, E–15
 - Tuning
 - parallel processing environment, 3–30 to 3–37
- Tutorial
 - See VAX C tutorial
- TYPE command (DEBUG), 2–10
- Type conversions, 7–13
- typedef** keyword, 8–32
- Type specifiers
 - discussion of, 8–3
 - summary of, D–4

U

- Unary expressions
 - address of, 7–11
 - cast, 7–13
 - increment and decrement, 7–10
 - indirection, 7–11
 - negation, 7–10
 - one's complement, 7–12
 - sizeof**, 7–14
- Unary operators
 - precedence of, 7–8
- /UNDEFINE CC qualifier, 1–10
 - examples and usage, 1–18
- #undef** preprocessor directive, 10–4
- union** keyword, 8–20
- Unions, 8–15
 - debugger access to, 2–28

Unions (cont'd.)

- declaration of, 8–20, 8–22
- initialization of, 8–26
- members of

 - references to, 7–6

- tutorial information, 4–22

- variant aggregates, 8–28

UNIX system environment, 4–2

\$UPDATE RMS function, 12–6

User account parameters

- tuning for parallel processing, 3–36

User-defined functions

- See functions

User-named psects, 14–2 to 14–5

User quotas

- DEBUG MP requirements, E–25 to E–26

- VAX C requirements, 3–36

Utilities

- LINT, 5–22

V

\v (vertical tab), 8–7

Vacuous tag declarations, 8–23

Values

- defined, 8–2

- of constants, 8–2

- of variables, 8–2

- tutorial information, 4–17

varargs functions and macros, 5–6

Variables

- as address expression for SET WATCH command (DEBUG), 2–18

- character, 8–4

- declarations

 - format of, 8–3

 - in overlapping blocks, 5–21

 - tutorial information, 4–5

- DEPOSIT command (DEBUG), 2–21

- EVALUATE command (DEBUG), 2–21

- EXAMINE command (DEBUG), 2–20

- global section (DEBUG MP), E–19

- identifiers, 5–17

- nonstatic

 - while debugging, 2–18, 2–20

- objects of, 8–2

- values of, 8–2

- watchpoint (DEBUG MP), E–19

variant_struct keyword, 8–28

variant_union keyword, 8–28

vax11c predefined macro, 11–4

VAXC\$INCLUDE logical, 10–17

VAX Calling Standard

- double** parameters, 5–12

- parameter lists for VAX C functions, 5–12

- parameter-passing mechanisms, 13–6

- structures as parameters, 5–12, 8–21

- structure sizes exemption for VAX C, 13–14

VAXCDEF.TLB, 1–6, 10–20

- tutorial information, 4–7

VAX C language

- See also, Portability concerns

- See also, VAX C Run-Time Library (RTL)

- keywords, 5–18

- list of operators, 7–6

- parallel-processing support, 3–1 to 3–37

- program structure, 5–1

- tutorial information, 4–3 to 4–26

VAXCPAR.OLB, 3–29

vaxc predefined macro, 11–4

VAX C Run-Time Library (RTL)

- definition modules, A–1

- linking against object modules, 1–27

- linking against shareable images, 1–29

- linking options explained, 1–26

- portability concerns, 10–16

- prototypes, 5–11

- tutorial information, 4–6

VAX C tutorial, 4–4 to 4–26

- addresses, 4–17

- aggregates, 4–21 to 4–26

- AND operator, 4–15

- arguments, 4–5

- arrays, 4–21

- blocks, 4–11

- break** statement, 4–13

- case sensitivity, 4–6

- character strings, 4–21

- comments, 4–4

- compiling and linking, 4–9

- compound statements, 4–11

- conditional execution, 4–10 to 4–15

- data types, 4–5

- definition modules, 4–7

- DIGITAL Command Language (DCL), 4–9

- do** statement, 4–14

- equality operator (==), 4–11

- executing a program, 4–9

- for** statement, 4–15

- function body, 4–5

- functions, 4–4

- getchar**, 4–10

VAX C tutorial (cont'd.)

- if statement, 4–10
- #include preprocessor** directive, 4–12
- input/output (I/O), 4–6
- language keywords, 4–6
- linking against RTL libraries, 4–6
- linking against RTL shareable images, 4–6
- linking a program, 4–9
- loop incrementing, 4–16
- loops, 4–14
- lvalue, 4–17
- macros, 4–12
- newline character, 4–8
- OR operator, 4–11
- parameters, 4–5
- pointers, 4–17 to 4–21
- return** statement, 4–5
- rvalue, 4–17
- scalars, 4–21 to 4–26
- storage classes, 4–5
- structures and unions, 4–22 to 4–26
- switch** statement, 4–12, 4–13
- values, 4–17
- variable declarations, 4–5
- VAX C RTL, 4–6
- VMS, 4–9
- VMS file extensions, 4–9
- VMS file names, 4–9
- void** functions, 4–5
- white space, 4–4
- vax predefined macro, 11–4
- VAX Text Processing Utility
 - See Editing
- VAXTPU
 - See Editing
- Visible process (DEBUG MP), E–3, E–9
- %VISIBLE_PROCESS logical (DEBUG MP), E–13
- VMS operating system
 - See also, Record Management Services (RMS)
 - RMS, 12–1
 - tuning for parallel processing, 3–30
- vms predefined macro, 11–4
- VMS Run-Time Library (RTL), 13–1
- void** keyword
 - as a function parameter, 5–5
 - as a function return type, 5–5
 - functions, 8–32
 - pointers, 8–13
 - tutorial information, 4–5
- void** pointers, 8–13
- volatile** modifier, 9–23

W

- /WARNINGS CC qualifier, 1–18
- Watchpoint (DEBUG), 2–18
 - nonstatic variable, 2–18
- Watchpoints (DEBUG MP), E–19
 - global section, E–19
- while** keyword
 - loop decomposition in parallel processing, 3–2
- while** statement, 6–9
- White space, 5–22
- Widening conventions
 - argument-conversion rules, 5–13
- Working set size
 - tuning for parallel processing, 3–37
- _WRITE_GPR built-in function, 11–21

X

- XAB RMS data structure, 12–6
- XOR bitwise operator (^), 7–17

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local DIGITAL subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	_____	Local DIGITAL subsidiary or approved distributor
Internal ¹	_____	SDC Order Processing - WMO/E15 <i>or</i> Software Distribution Center Digital Equipment Corporation Westminster, Massachusetts 01473

¹For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

Guide to VAX C
AA-L370D-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)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Completeness (enough information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Clarity (easy to understand)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization (structure of subject matter)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Figures (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Examples (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Index (ability to find topic)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Page layout (easy to find information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

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 _____

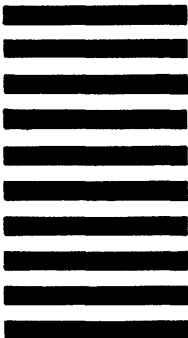
Phone _____

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

Guide to VAX C
AA-L370D-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)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Completeness (enough information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Clarity (easy to understand)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization (structure of subject matter)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Figures (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Examples (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Index (ability to find topic)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Page layout (easy to find information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

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 _____

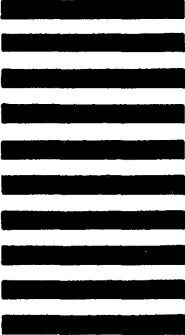
Mailing Address _____
_____ Phone _____

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

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

