Guide to VAX C

Order Number: AI–L370C–TE

March 1987

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 document supersedes *Programming in VAX C*, (Order No. AA–L370B–TE).
Operating System and Version:	VMS Version 4.2 or higher, or MicroVMS Version 4.2 or higher
Software Version:	VAX C Version 2.3

digital equipment corporation
maynard, massachusetts

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

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

ZK3222

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a trademark of the American Mathematical Society.

Contents

VAX C PROGRAMMING CONCEPTS

USING VAX C FEATURES ON VMS

FIGURES

TABLES

Preface

This manual 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 manual also includes information concerning the porting of 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 additional information concerning porting programs to and from other operating systems, refer to the *VAX C Run-Time Library Reference Manual.*

Intended Audience

This manual 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. Readers should be familiar with one high-level language. Readers should have some familiarity with the DIGITAL Command Language (DCL); those who do not, or those who need to reference information concerning DCL, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

Structure of This Document

This manual has 11 chapters and 4 appendixes. These chapters are grouped into three parts as follows:

Developing VAX C Programs on VMS

- Chapter 1, Developing VAX C Programs at DCL Command Level, explains how to edit, compile, link, and run a VAX C program.
- Chapter 2, Using the VMS Debugger, explains how to use the VMS Debugger.

VAX C Programming Concepts

- Chapter 3, Program Structure, explains program structure.
- Chapter 4, Statements, describes VAX C statements.
- Chapter 5, Expressions and Operators, discusses expressions and operators used in VAX C.
- Chapter 6, Data Types and Declarations, explains data types and declarations.
- Chapter 7, Storage Classes and Allocation, describes storage classes and allocation.
- Chapter 8, Preprocessor Directives, explains preprocessor directives.

Using VAX C Features on VMS

- Chapter 9, Using VAX Record Management Services (RMS) explains VAX Record Management Services (RMS).
- Chapter 10, Using VAX C in the Common Language Environment, describes System Services and Run-Time Library routines.
- Chapter 11, VAX C Implementation Notes, explains VAX C implementations.

Appendixes

- Appendix A, VAX C Definition Modules, describes VAX C definition modules.
- Appendix B, VAX C Compiler Messages, lists VAX C compiler messages.
- Appendix C, Optional Programming Productivity Tools, provides an overview of the VAX Language-Sensitive Editor (VAXLSE) as used in conjuction with the VAX Source Code Analyzer (VAXSCA).
- Appendix D, VAX C Language Summary, provides a summary of all VAX C language 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 additional information concerning porting programs to and from other operating systems.
- *VMS Master Index*—For programmers who need to work with the VAX machine architecture or the VMS System Services. This index lists manuals that cover the individual topics concerning access to VMS.
- *The C Programming Language*[1] —For those who need a more intensive tutorial than that provided in Chapter 3, Program Structure. VAX C contains features and enhancements to the C language as defined in *The C Programming Language*. Therefore, the *Guide to VAX C* should be used as the reference book for the full description of VAX C.

Conventions Used in This Document

Convention	Meaning
RETURN	The symbol RETURN represents a single stroke of the RETURN key on a terminal.
CTRL/X	The symbol CTRL/X, 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.

[1] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Englewood Cliffs, New Jersey: Prentice-Hall, 1978).

Convention	Meaning
$ RUN CPROG RETURN	In interactive examples, the user's response to a prompt is printed in red; system prompts are printed in black.
float x; . . . x = 5;	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.
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 a multidimensional 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	Boldface type identifies 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 2.3 from previous versions:

- VAX C now supports VAXSCA. See Appendix A, VAX C Definition Modules, for more information.
- You cannot link modules created by the VAX C V1.n compilers with modules created by the VAX C V2.n compilers. You must recompile your code.
- The VAX C Run-Time Library (RTL) is no longer distributed with VAX C and is now distributed with the VMS Run-Time Library.
- VAX C allows you to use function prototypes in function declarations. To use prototypes, you specify the data type of the arguments in a function declaration. The compiler checks the argument specifications in the prototype against the parameters in the call and provides error checking and necessary data type conversions. For more information, refer to Chapter 3, Program Structure.
- You should not use the underscore as the first character in program identifiers since VAX C uses the underscore to identify implementation-specific constants, macros, and keywords. For more information, refer to Chapter 3, Program Structure.
- VAX C no longer performs all floating-point arithmetic in double precision. VAX C uses the precision of the operand with the highest precision if /PRECISION=SINGLE is specified. For more information, refer to Chapter 5, Expressions and Operators.
- VAX C supports the **volatile** and **const** data type specifiers. For more information, refer to Chapter 6, Data Types and Declarations.
- VAX C supports the /INCLUDE_DIRECTORY qualifier, which provides an additional level of search for user-defined include files.
- VAX C supports variant structure and union declarations nested within structure or union declarations. For more information, refer to Chapter 6, Data Types and Declarations.
- VAX C supports vacuous tag declarations that eliminate ambiguity in forward references to structure and union tags. For more information, refer to Chapter 6, Data Types and Declarations.

- VAX C supports hexadecimal characters in escape sequences. For more information, refer to Chapter 6, Data Types and Declarations.
- VAX C provides a way for you to align data on byte, word, longword, quadword, octaword, or page boundaries. For more information, refer to Chapter 7, Storage Classes and Allocation.
- VAX C supports the **#pragma** preprocessor directive. For more information, refer to Chapter 8, Preprocessor Directives.
- VAX C supports the **#elif** preprocessor directive. For more information, refer to Chapter 8, Preprocessor Directives.
- VAX C supports the **defined** operator in the **#if** preprocessor directive. For more information, refer to Chapter 8, Preprocessor Directives.
- VAX C includes the predefined macros _align, __DATE__, __FILE__, __LINE__, and __TIME__. For more information, refer to Chapter 8, Preprocessor Directives.
- VAX C allows macro substitution within the **#include** preprocessor directives. For more information, refer to Chapter 8, Preprocessor Directives.
- VAX C includes the new library files limits.h and float.h. For more information, refer to Appendix A, VAX C Definition Modules.
- VAX C provides many new diagnostic messages. For more information, refer to Appendix B, VAX C Compiler Messages.

veloping VAX C Programs on VMS

Chapter 1

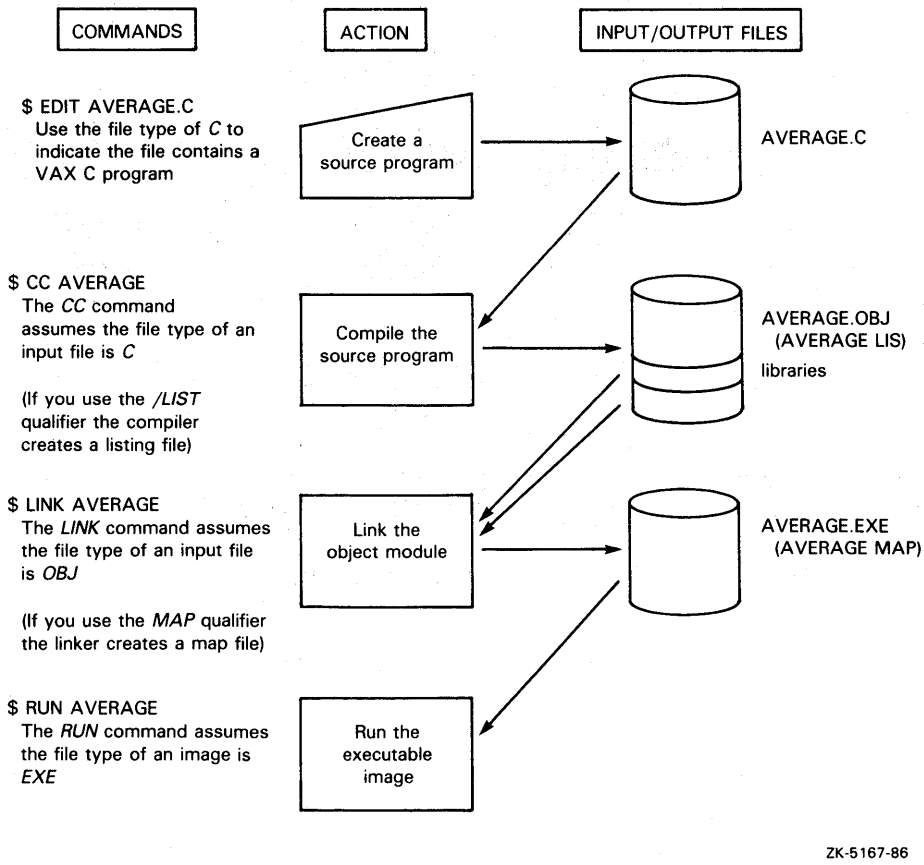
Developing VAX C Programs at DCL Command Level

This chapter describes how to create, compile, link, and run a VAX C program using DCL commands.

1.1 DCL Commands for Program Development

This section briefly describes the DCL commands that are used to create, compile, link, and run a VAX C program on a VMS system. These commands are shown in Figure 1–1. For a more detailed description of each command, see the sections that follow.

Figure 1–1: DCL Commands for Developing Programs



The following example shows each of the commands shown in Figure 1–1 executed in sequence.

```
$ EDIT/EDT FIRST_PROG.C
$ CC FIRST_PROG
$ LINK FIRST_PROG
$ RUN FIRST_PROG
```

To create a VAX C source program at DCL level, you must invoke a text editor. In the previous example, the VAX EDT editor is invoked to create the source program FIRST_PROG.C. You can, however, use another editor, such as the VAX Text Processing Utility (VAXTPU) or the VAX Language-Sensitive Editor (VAXLSE). 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; VAX C searches for C by default.

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 a listing file.

```
$ CC/LIST FIRST_PROG
```

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

Once your program has compiled successfully, you invoke the VMS Linker to create an executable image file. The VMS Linker uses the object file produced by VAX C as input to produce an executable image file as output.

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.

Once the executable image file has been created, you can run your program with the DCL command RUN.

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 sections describe briefly how to use both EDT and VAXTPU.

1.2.1 Using VAX EDT

EDT is an interactive general-purpose text editor that offers three editing modes: keypad, nokeypad, and line. With keypad mode, you issue commands by using the numeric keypad that appears to the right of your main keyboard. With nokeypad mode, you enter commands on a command line, which EDT processes when you press the RETURN key. Line mode focuses on the line as the unit of text. With line mode, you issue commands at the line mode asterisk prompt (*).

Keypad mode and nokeypad mode continually display the contents of the file on your screen. When you begin your editing session, editing in line mode is the default. Unlike keypad and nokeypad mode, line mode displays only one line of text on your screen.

The following command line invokes the EDT editor and creates the file PROG_1.C.

```
$ EDIT/EDT PROG_1.C
```

To change from line mode to keypad mode, type the CHANGE command at the asterisk prompt. To return to line mode from keypad mode, press CTRL/Z. To change from line mode to nokeypad mode, type the SET NOKEYPAD command and then type the CHANGE command.

When you invoke EDT to create a file, a journal file is created automatically. You can use this journal file to recover your edits if the system fails during an editing session. To recover your edits, type the EDIT/RECOVER command.

EDT provides an online HELP facility that you can access during an editing session. In line mode, you can type the HELP command. EDT displays general information on EDT as well as detailed information on both line mode editing and nokeypad mode editing. In keypad mode, you can press the HELP key or the PF2 key. EDT displays a keypad diagram on your screen and a list of keypad editing keys. For help on a specific keypad function, press the key you want help on.

For more detailed information on how to use EDT, see the *VAX EDT Reference Manual.*

1.2.2 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 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 created automatically. 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.

1.2.2.1 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"
```

Once 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 Text Processing on VAX/VMS.*

1.2.2.2 The EDT Keypad Emulator Interface

The EDT Keypad Emulator interface provides all of the functions associated with EDT and uses the same keys to perform each function. The following command line invokes the EDT Keypad Emulator:

```
$ EDIT/TPU/SECTION=EDTSECINI.GBL
```

You can define a global symbol by placing a symbol definition in your LOGIN.COM file. For example:

```
$ EDTEM == "EDIT/TPU/SECTION=EDTSECINI"
```

When this command line is executed, you can type EDTEM at the DCL prompt followed by the name of the file you want to create or modify. For example:

```
$ EDTEM PROG_1.C
```

For more detailed information on how to use the EDT Keypad Emulator, see the *VAX Text Processing Utility Reference Manual.*

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 VMS 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 files listed. However, when a qualifier appears after a file specification, it affects only the file that immediately precedes it. When files are concatenated, however, 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; 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. For instance, 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
```

Note that 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.

When 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. These modules are identical to the set of .H files that are supplied with VAX C. 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 either of these **#include** lines:

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

which includes the file SYS$LIBRARY:STDIO.H; or equivalently

```
#include stdio
```

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.

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.

The CC command qualifiers are as follows:

Command Qualifiers	Default
/[NO]ANALYSIS_DATA[=file-spec]	/NOANALYSIS_DATA
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG[=(option,...)]	/DEBUG=TRACEBACK
/[NO]DEFINE[=(definition list)]	/NODEFINE
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS
/[NO]G_FLOAT	/NOG_FLOAT
/[NO]INCLUDE_DIRECTORY=(pathname [,...]) /LIBRARY	/NOINCLUDE_DIRECTORY
/[NO]LIST[=file-spec]	/NOLIST (interactive mode) /LIST (batch mode)
/[NO]MACHINE_CODE[=option]	/NOMACHINE_CODE
/[NO]OBJECT[=file-spec]	/OBJECT
/[NO]OPTIMIZE[=NODISJOINT]	/OPTIMIZE

/[NO]PRECISION={SINGLE,DOUBLE}	/PRECISION=DOUBLE
/SHOW[=(option, . . .)]	/SHOW=(NOBRIEF, NODICTIONARY, NOEXPANSION, NOINCLUDE, NOINTERMEDIATE, NOSTATISTICS, NOSYMBOLS, NOTRANSLATION, SOURCE, TERMINAL)
/[NO]STANDARD[=(option,...)	/NOSTANDARD
/[NO]UNDEFINE[=(undefine list)]	/NOUNDEFINE
/[NO]WARNINGS[=(option,...)]	/WARNINGS

Command qualifiers can be placed either on the CC command 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, 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 command qualifiers.

/[NO]ANALYSIS_DATA[=files-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.

/[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.

The default is /NOCROSS_REFERENCE.

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

Performs, from the command line, the same functions performed by the **#define** and **#undefine** preprocessor directives. That is, /DEFINE defines a token string or macro to be substituted for every occurrence of a given identifier in the compilation unit(s); /UNDEFINE cancels a

previous definition. When both /DEFINE and /UNDEFINE are present in a compilation unit or on the CC command, /DEFINE is evaluated before /UNDEFINE.

All command qualifiers are first processed by DCL. The command line printed at the end of a listing file shows the result of this translation. Since the 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 list illustrates particular 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 or in a single compilation unit, only the last /DEFINE is used. Similarly, only the last /UNDEFINE is used on the CC command or the compilation unit.

The /DEFINE qualifier is used to define identifiers for constant expressions or to define preprocessor macros. The simplest form of a /DEFINE definition is

```
/DEFINE=true
```

This results in a definition like the one that would result from

```
#define  TRUE  1
```

Macro definitions must be enclosed in quotation marks. DCL cannot recognize a definition of the following form:

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

and issues a warning message. The correct definition is written as follows:

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

This definition produces the same results as

```
#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

```
#define  true  1
```

However, the definition

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

is equivalent to

```
#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

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

However, the definitions

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

are not recognized by DCL. 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(""%d\n"",b)")
```

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

Any definition specified by a /DEFINE qualifier can be cancelled by the /UNDEFINE qualifier by giving the name of the identifier or macro. For example:

```
$ CC/UNDEFINE="quotes"
```

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. Given 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
%CC-W-UNDEFIFMAC, "vaxc" is not a currently defined macro;
                  constant zero assumed.
                  At line number 3 in DBA0:[MADRIGAL]EXAMPLE.C;3.

%CC-I-SUMMARY, Completed with 0 error(s), 1 warning(s), and
                  0 informational messages.
                  At line number 8 in DBA0:[MADRIGAL]EXAMPLE.C;3.

$ LINK  EXAMPLE.OBJ RETURN
%LINK-W-WRNERS, compilation warnings
    in module EXAMPLE file DBA0:[MADRIGAL]EXAMPLE.OBJ;4

$ RUN  EXAMPLE.EXE RETURN
I'm being compiled on some other compiler.
```

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 printed before the source listing and in the order in which 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 become quite long. Continuation characters cannot appear within quotes or they will be included in the token stream. Note, too, that the length of a command line cannot exceed the maximum length allowed by DCL.

The defaults are /NODEFINE and /NOUNDEFINE.

/[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 diagnostics file is reserved for use with DIGITAL layered products.

The default is /NODIAGNOSTICS.

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

Requests information to be included in the object module for use by the VMS Debugger. You may select one or more of the following options:

Option	Usage
ALL	Includes symbol table records and traceback records. This is equivalent to /DEBUG with no option.
NONE	Does not include any debugging information. This is equivalent to /NODEBUG.
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 *not* 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.

/[NO]G_FLOAT

Controls the format of floating-point variables. When you do not specify /G_FLOAT on the CC command line, **double** variables are represented in D_floating format. When /G_FLOAT is specified, all variables declared as **double** are represented in G_floating format. A program compiled with /G_FLOAT must also be linked with the G_floating library,

VAXCRTLG.OLB. This library must be specified so that it is searched before VAXCRTL.OLB. For more information concerning the G_floating attribute, refer to Chapter 6, Data Types and Declarations.

The default is /NOG_FLOAT.

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

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

The /INCLUDE_DIRECTORY qualifier is meant to provide the functionality of the -i qualifier in PCC 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 source file.
2. The directories specified in the /INCLUDE 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 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 SYS$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 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. 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. However, in 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
```

For more information on the use of text libraries, refer to Section 1.3.1.

/[NO]LIST[=file-spec]

Directs the compiler to produce a listing file. You must specify this qualifier whenever you need any type of listing. None of the other qualifiers use /LIST.

When /LIST is in effect, the compiler, by default, creates 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.

/[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.

There exist several formats for the listing of machine code. You may select one of the following options.

Option	Usage
AFTER	The option AFTER causes the lines of machine code produced during compilation to print after all of the source code in the listing.
BEFORE	The option BEFORE causes lines of machine code produced during compilation to print before any source code in the listing.
INTERSPERSED	The option 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]PRECISION= { SINGLE / DOUBLE }

Controls whether floating-point operations on float variables and constants are performed in single or double precision.

The default is /PRECISION=DOUBLE.

/[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 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 concerning file specifications.

The compiler executes more rapidly 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.

/[NO]OPTIMIZE[=NODISJOINT]

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 three-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.

The default is /OPTIMIZE.

/SHOW=(option, . . .)

The qualifier /SHOW sets or cancels listing options. You must use the /LIST qualifier with the /SHOW qualifier to select or cancel any of the following options:

Option	Usage
ALL	The option ALL prints all listing information.
[NO]BRIEF	The option BRIEF prints the same listing as the option SYMBOLS except that BRIEF will eliminate from the list any identifiers that are not actually referenced in the program and are not members of a structure or union that is referenced in the program. The option /NOBRIEF is the default.
[NO]DICTIONARY	The option DICTIONARY prints the Common Data Dictionary definitions included in the program with the **#dictionary** preprocessor directive. Note that these data definitions are marked in the listing file with an uppercase letter D in the listing margin. The option NODICTIONARY is the default.
[NO]EXPANSION	The option EXPANSION prints 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 option NOEXPANSION is the default.
[NO]INCLUDE	The option INCLUDE prints the contents of **#include** files and modules in the program listing. The option NOINCLUDE is the default.
[NO]INTERMEDIATE	The option INTERMEDIATE prints all intermediate and also the final macro expansions in the program listing. The option NOINTERMEDIATE is the default.

Option	Usage
NONE	The option NONE prints an empty listing file, with only the header. If you specify this option on a command line that contains /LIST and /MACHINE_CODE, the compiler places machine code in the list.
[NO]SOURCE	The option SOURCE prints the source program statements in the program listing. The option SOURCE is the default.
[NO]STATISTICS	The option STATISTICS prints compiler performance statistics in the program listing. The option NOSTATISTICS is the default.
[NO]SYMBOLS	The option SYMBOLS prints 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 option NOSYMBOLS is the default.
[NO]TERMINAL	The option TERMINAL displays compiler messages to the terminal. The option TERMINAL is the default.
[NO]TRANSLATION	The option TRANSLATION prints all UNIX system file specifications that the compiler translates to VMS file specifications using DEC/Shell functions. For more information on file translation, refer to the *VAX C Run-Time Library Reference Manual.* The option NOTRANSLATION 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 any 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]WARNINGS[=(option,...)]

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

The two options are as follows:

Option	Usage
NOINFORMATIONALS	The option NOINFORMATIONALS causes the compiler to suppress informational messages.
NOWARNINGS	The option 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.

1.3.3 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. You should 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
```

The parts of this message are described in the following list:

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

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

message-text
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*
The integer *m* gives 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*
The integer *n* gives 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*. The **#line** directive can be used to change both the line number and name that appear in the message.

The messages produced by the VAX C compiler are listed in Appendix B, VAX C Compiler Messages.

Both the CC command and the DCL command SET MESSAGE give you control over the display of messages. The CC qualifier /NOWARNINGS, discussed previously, suppresses warning messages generated by the compiler. The DCL command SET MESSAGE lets you decide whether messages will be displayed in their entirety or in a shortened form. For example, if you do not want to see the %CC-s-ident part of messages, you can enter the command:

```
$ SET  MESSAGE/NOFACILITY/NOSEVERITY/NOIDENTIFICATION RETURN
```

This command cancels the facility, severity, and identification portion of all messages. It remains in effect for all commands you subsequently enter, until you reissue the SET MESSAGE command or log out of the system.

1.3.4 Compiler Listings

A compiler listing provides information that can help you debug your VAX C program. To generate a listing file, specify the /LIST qualifier when you compile your VAX C program interactively. For example:

```
$ CC/LIST
```

If the program is compiled in batch mode, the listing file is created by default; specify the /NOLIST qualifier to suppress creation of the listing file. (In either case, the listing file is not printed automatically.) By default, the name of the listing file is the name of the source program with a file type of LIS. You can include a file specification with the /LIST qualifier to override this default.

A compiler listing generated by the /LIST qualifier has the following sections:

- Source Program Listing

  This section displays the source code plus line numbers generated by the compiler.

- Storage Map

  This section displays summary information on program sections, variables, and arrays.

- Compilation Summary

  This section displays the qualifiers used with the CC command and the compilation statistics.

When used with the /LIST qualifier, the following CC command qualifiers supply additional information in the compiler listing:

- /CROSS_REFERENCE
- /MACHINE_CODE
- /SHOW

See Section 1.3.2 for a description of each qualifier's function.

When the CC command line contains the /LIST qualifier but does not contain the /SHOW qualifier, you are given the default listing. All the information in the default listing is also included in the other listing formats. The default listing includes:

- Margin information
- The VAX C source text
- Any errors encountered during the compilation
- The command line used to invoke the compiler

The left-hand margin of the source listing produced by the VAX C compiler contains several items of information, arranged into fields in the following format:

```
nnnnn  i  ss  mm
```

nnnnn
is the compiler-generated listing line number; it starts at 1, and is incremented by one for each line in the source program, including lines read from included files (whether or not the /SHOW=INCLUDE qualifier was specified in the command line).

i
is the level of nesting of lines read from included files; this field is present only if /SHOW=INCLUDE was specified on the command line. Level 0, which appears as a blank, indicates lines read from the source file, or files, specified on the command line.

ss
is the level of nesting of compound statements in the source program; it starts at zero (which appears as a blank) for external definitions and declarations, is incremented each time a left brace, which introduces a compound statement, is encountered (including the brace which introduces a function body), and is decremented at the corresponding right brace. This field may also appear as an "X" instead of a number; this indicates that the source line is being ignored by the compiler as a result of the evaluation of a previous **#if**, **#ifdef**, or **#ifndef** preprocessor directive. This field may also appear as a "D" instead of a number; this indicates that the compiler has replaced the **#dictionary** preprocessor directive, and the specified Common Data Dictionary data structure, with VAX C source code. The flagging of CDD replacements only occurs if you specify /SHOW=DICTIONARY on the CC command line.

mm
is the level of nesting of the last macro expanded in the line; this field is present only if the qualifier /SHOW=EXPANSIONS or /SHOW=INTERMEDIATE was specified on the command line. Level 0 corresponds to the original source line, and appears as a blank. When this field is nonzero, however, the fields "nnnnn", "i", and "ss" all appear as blanks.

In all cases, the numbers listed are right-justified in their fields, with no leading zeros.

NOTE

> The spacing within the compiler listings in this appendix may not be consistent with the spacing in actual compiler listings. The listings in this index are condensed so as to fit on the printed page.

Example 1–1 shows the default compiler listing. The /STANDARD=PORTABLE option was used to show how the compiler lists error messages.

Example 1–1: Default Compiler Listing

```
❶ EXAMPLE    14-JAN-1985 14:07:31    VAX C     V2.0-00  Page 1
❷ V1.0       14-JAN-1985 14:02:10    DBA0:[.D]EXAMPLE.C;2  (1)

❸     1          /*  This is a sample program to show the  *
      2           *  format of the compiler listing, as    *
      3           *  well as the effect of the various     *
      4           *  /SHOW command-line qualifier values.  */
      5
      6          /*  This line shows what happens when a line
  from the source file exceeds the right listing margin. */
      7
      8          #include "debugging.h"
     13
     14          #include timeb

❹ %CC-W-INCMODNOTPORT, #include of a library module is not
                       portable.

     32
     33          #define NULL   0
     34
     35          main()
     36          {
❺    37    1       globalvalue  SS$NORMAL;

  %CC-W-NONPORTSC, "globalvalue" is a nonportable storage
                   class specifier.

     38    1       struct       timeb  time_struct;
     39    1       char         *ctime_string, *ctime();
     40    1       int          status = SS$NORMAL;
     41    1
     42    1       ftime (& time_struct);
     43    1       if ((ctime_string =
     44    1            ctime(& time_struct.time))
     45    1            != NULL)
     46    1
```

(Continued on next page)

Example 1–1 (Cont.): Default Compiler Listing

```
47    1            printf ("Run time is %s",
48    1                    ctime_string);
49    1
50    1       #if debugging
51    X          printf ("\n*** Debugging version \
52    X       ***\n\n");
53    X
54    X          status =
55    1       #endif
56    1
57    1          process();
58    1          return status;
59    1       }


  Command Line
  ------------

❻ CC/LIST/STANDARD=PORTABLE  EXAMPLE.C
```

Key to Example 1–1:

❶ The name of the module and its identification appear at the top left of the listing. The date and time of compilation and the version of the compiler used appear at the center of the listing.

❷ The date and time when the source file was created and the file specification appear below the information to the right in number ❶. The page numbers of the listing file and the source file (in parentheses) are to the far right of the listing.

❸ The compiler generates the listing line numbers.

❹ Diagnostic messages appear immediately following the source line in question.

❺ The nesting level of compound statements starts at zero (appearing as blanks), is incremented each time a left brace is encountered, or appears as an "X", which means that the line is ignored after a conditional directive is evaluated.

❻ The command line that generated the listing appears at the bottom.

The CC command line qualifiers /SHOW=EXPANSIONS and /SHOW=INTERMEDIATE cause the compiler listing to display the results of macro substitution. The qualifier /SHOW=EXPANSIONS displays only the final, fully-substituted line, immediately following the listing of the original source line. The qualifier /SHOW=INTERMEDIATE displays the complete progress of substitution, printing a new line each time the compiler replaces a macro reference by its definition. If the compiler issues an error message against a line which contains substitutions, the message appears between the original line and the first substituted line.

The purpose of displaying the results of macro substitution is to show the source line as it is ultimately seen by the compiler before translation to object code.

Example 1–2 is a listing which shows **#include** modules and intermediate macro expansions.

Example 1–2: Listing Format of Macro Substitutions

```
EXAMPLE    14-JAN-1985 14:07:31    VAX C     V2.0-00  Page 1
V1.0       14-JAN-1985 14:02:10    DBA0:[.D]EXAMPLE.C;2  (1)

     1          /*  This is a sample program to show the  *
     2           *  format of the compiler listing, as    *
     3           *  well as the effect of the various     *
     4           *  /SHOW command-line qualifier values.  */
     5
     6          /*  This line shows what happens when a line
from the source file exceeds the right listing margin. */
     7
     8          #include "debugging.h"
❶    9 1        #define  YES       1
    10 1        #define  NO        0
    11 1        #define  debugging NO
    12 1
    13 1        /*  Yes only if debugging code is *
    14 1         *   to be compiled               */
    15
    16          #include timeb
    17 1        /*     TIMEB - Ftime() RTL Routine Return
                                Structure Definition  */
    18 1
    19 1        #include types
❷   20 2        /*   TYPES - RTL Typedef Definitions */
    21 2
    22 2        #ifndef TYPES_H_DEFINED
    23 2        typedef long int time_t;
    24 2        #define TYPES_H_DEFINED
    25 2        #endif
    26 1
    27 1        struct timeb
    28 1           {
    29 1              time_t          time;
    30 1              unsigned short  millitm;
    31 1              short           timezone;
    32 1              short           dstflag;
    33 1           };
    34
```

(Continued on next page)

Example 1–2 (Cont.): Listing Format of Macro Substitutions

```
    35          #define NULL   0
    36
    37          main()
    38          {
    39    1        globalvalue  SS$NORMAL;
    40    1        struct       timeb  time_struct;
    41    1        char         *ctime_string, *ctime();
    42    1        int          status = SS$NORMAL;
    43    1
    44    1        ftime (& time_struct);
    45    1        if ((ctime_string =
    46    1             ctime(& time_struct.time))
❸   47    1            != NULL)
          1             != 0)
    48    1
    49    1           printf ("Run time is %s",
    50    1                   ctime_string);
    51    1
    52    1     #if debugging
          1     #if NO
          2     #if 0
    53    X        printf ("\n*** Debugging version \
    54    X     ***\n\n");
    55    X
    56    X        status =
    57    1     #endif
    58    1
    59    1        process();
    60    1        return status;
    61    1     }

Command Line
------------

CC/LIST/SHOW=(INCLUDE,INTERMEDIATE,EXPANSION) EXAMPLE.C
```

Key to Example 1–2:

❶ The level of nesting of lines read from **#include** files is shown. This column is blank when the lines are read from the source file.

❷ The nesting level is incremented when the **#include** file also contains **#include** directives.

❸ The level of nesting of the last macro expanded in the line is shown.

When you use the /CROSS_REFERENCE option, the compiler produces a storage map with symbol table cross-references, as shown in Example 1–3. The storage map produced by the /SHOW=SYMBOLS is the same as the listing shown in Example 1–3, except that the cross-references are not included.

Example 1–3: Cross-Reference Listing

```
EXAMPLE    14-JAN-1985 14:07:31    VAX C     V2.0-00  Page 1
V1.0       14-JAN-1985 14:02:10    DBA0:[.D]EXAMPLE.C;2  (1)

    1           /*  This is a sample program to show the  *
    2            *  format of the compiler listing, as    *
    3            *  well as the effect of the various     *
    4            *  /SHOW command-line qualifier values.  */
    5
    6           /*  This line shows what happens when a line
from the source file exceeds the right listing margin.  */
    7
    8           #include "debugging.h"
    9 1         #define  YES       1
   10 1         #define  NO        0
   11 1         #define  debugging NO
   12 1
   13 1         /*  Yes only if debugging code is *
   14 1          *   to be compiled               */
   15
   16           #include timeb
   17 1         /*   TIMEB - Ftime() RTL Routine Return
                                Structure Definition  */
   18 1
   19 1         #include types
   20 2         /*   TYPES - RTL Typedef Definitions */
   21 2
   22 2         #ifndef TYPES_H_DEFINED
   23 2         typedef long int time_t;
   24 2         #define TYPES_H_DEFINED
   25 2         #endif
   26 1
   27 1         struct timeb
   28 1            {
   29 1               time_t            time;
   30 1               unsigned short    millitm;
   31 1               short             timezone;
   32 1               short             dstflag;
   33 1            };
   34
```

(Continued on next page)

Example 1–3 (Cont.): Cross-Reference Listing

```
35          #define NULL   0
36
37          main()
38          {
39    1        globalvalue  SS$NORMAL;
40    1        struct       timeb  time_struct;
41    1        char         *ctime_string, *ctime();
42    1        int          status = SS$NORMAL;
43    1
44    1        ftime (& time_struct);
45    1        if ((ctime_string =
46    1             ctime(& time_struct.time))
47    1             != NULL)
        1            != 0)
48    1
49    1           printf ("Run time is %s",
50    1                   ctime_string);
51    1
52    1       #if debugging
          1     #if NO
          2     #if 0
53    X           printf ("\n*** Debugging version \
54    X       ***\n\n");
55    X
56    X           status =
57    1       #endif
58    1
59    1           process();
60    1           return status;
61    1       }

                 +-------------+
                 | Storage Map |
                 +-------------+
```

❶ External Declarations

```
---------------------

Identifier
   Name      Line     Size       Class     Type and References
----------   ----     ----       -----     -------------------
```

❷ main — Line 37 — Class: Extern def. — Function returning long int — No references

(Continued on next page)

Example 1–3 (Cont.): Cross-Reference Listing

```
timeb          27    10 bytes              Structure tag
                                           - Referenced at
                                             line 40

time           29    1 longword            Member(offset = 0),
                                           long int
                                           - Referenced at
                                             line 46

millitm        30    1 word                Member (offset =
                                                  4 bytes),
                                           unsigned short int
                                           - No references

timezone       31    1 word                Member (offset =
                                                  6 bytes),
                                           short int
                                           - No references

dstflag        32    1 word                Member (offset =
                                                  8 bytes),
                                           short int
                                           - No references

time_t         23    1 longword            Typedef:  long int
                                           - Referenced at
                                             line 29
```

❸ `Function "main" defined at line 37`

```
                                  ---------------------------------

Identifier
   Name      Line     Size       Class    Type and References
----------   ----     ----       -----    -------------------

ctime          41               Extern    Function returning
                                          pointer to char
                                          - Referenced at
                                            line 46

ctime_string   41   1 longword   Not      Pointer to char
                                Alloc.    - Referenced at
                                            lines 45 and 50

ftime          44               Extern    Function returning
                                          long int
                                          - Referenced at
                                            line 44
```

(Continued on next page)

Example 1–3 (Cont.): Cross-Reference Listing

```
printf          50                Extern    Function returning
                                            long int
                                            - Referenced at
                                              line 50
process         59                Extern    Function returning
                                            long int
                                            - Referenced at
                                              line 59

SS$NORMAL       39  1 longword    Global    Long int
                                            - Referenced at
                                              line 42

status          42  1 longword    Register  Initialized long int
                                            - Referenced at
                                              line 60

time_struct     40  10 bytes      Auto      Struct timeb
                                            - Referenced at
                                              lines 44 and 46
```

❹ Psect Synopsis

```
---------------

Psect Name                  Allocation  Attributes
----------                  ----------  ----------

$CODE                       68 bytes    Position-independent,
                                        relocatable,
                                        shareable,
                                        executable, readable

$CHAR_STRING_CONSTANTS      15 bytes    Position-independent,
                                        relocatable,
                                        readable, writeable
```

❺ Function Definition Map

```
-----------------------

 Line   Name
 ----   ----

37      main

Command Line
------------

CC/LIST/SHOW=(INCLUDE,INTERMEDIATE,SYMBOLS)-
  /CROSS_REFERENCE  EXAMPLE.C
```

Key to Example 1–3:

❶ The External Declarations section of the Storage Map lists all names declared or defined outside of any function.

❷ When the /CROSS_REFERENCE option is used, the compiler gives the line number in which each name is referenced. The cross-reference information is not included in the storage map unless the /CROSS_REFERENCE option is used.

❸ For each function in the source program, the compiler lists each declared name, giving:

- The identifier of the name
- The line on which the name is declared
- The size of the identifier
- The storage class to which the name belongs
- The data type of the name

❹ The Program Section (psect) Synopsis lists the program sections created by the compiler and their attributes.

❺ The Function Definition Map lists each function defined in the program and gives the line number in which the function is defined.

When you use the /SHOW=STATISTICS option, the compiler accumulates and displays statistics for each phase of its operation. It then lists the amount of I/O, memory, and CPU time used during the compilation. Example 1–4 shows the information returned by the /SHOW=STATISTICS option.

Example 1–4: Compiler Performance Statistics

```
EXAMPLE    14-JAN-1985 14:07:31    VAX C     V2.0-00  Page 1
V1.0       14-JAN-1985 14:02:10    DBA0:[.D]EXAMPLE.C;2  (1)

    1           /*  This is a sample program to show the  *
    2            *  format of the compiler listing, as    *
    3            *  well as the effect of the various     *
    4            *  /SHOW command-line qualifier values   */
    5
    6           /*  This line shows what happens when a line
from the source file exceeds the right listing margin. */
    7
    8           #include "debugging.h"
   15
   16           #include timeb
   34
   35           #define NULL   0
   36
   37           main()
   38           {
   39    1         globalvalue  SS$NORMAL;
   40    1         struct       timeb  time_struct;
   41    1         char         *ctime_string, *ctime();
   42    1         int          status = SS$NORMAL;
   43    1
   44    1         ftime (& time_struct);
   45    1         if ((ctime_string =
   46    1              ctime(& time_struct.time))
   47    1              != NULL)
   48    1
   49    1            printf ("Run time is %s",
   50    1                     ctime_string);
   51    1
   52    1      #if debugging
   53    X         printf ("\n*** Debugging version \
   54    X      ***\n\n");
   55    X
   56    X         status =
   57     1     #endif
   58     1
   59     1        process();
   60     1        return status;
   61     1     }
```

(Continued on next page)

Example 1–4 (Cont.): Compiler Performance Statistics

```
Command Line
------------

CC/LIST/SHOW=STATISTICS  EXAMPLE.C

                +------------------------+
                | Performance Indicators |
                +------------------------+

❶ phase           buf    dir   page  virt
                  i/o    i/o   flt   mem    workset  cputim
-----             -----  ----- ----  ----   -------  ------
  parse/semantics
  totals            7     13   148    64     1200     102

❷  live analysis    0      0    11     0     1200       4

   reorder
   invariants       0      0     4     0     1200       1

   eliminate
   redundancy       0      0     3     0     1200       2

❸ optimizer totals  0      0    26     0     1200      13

  allocator totals  0      0     0     0     1200       1

   generate code
   list             0      0    18     0     1200      10

   register
   allocation       0      0     0     0     1200       1

   peephole
   optimization     0      0     3     0     1200       3

   branch/jump
   resolution       0      0     3     0     1200       1

   write object
   module           0      0     6     0     1200       1

  code generator
  totals            0      0    35     0     1200      20

❹ total compilation 10     18   256    64     1200     166

  61 lines compiled

❺ compilation rate was 2204 lines per minute
```

Key to Example 1–4:

❶ The cputim column shows the maximum working set size, not the total.

❷ The subphases of the compiler are indented in this column and precede the totals for their phase.

❸ The phase totals follow the breakdown of their subphases.

❹ The totals for the performance indicators may be greater than the sum of the phase totals. For example, the buffered I/O total (buf I/O) is 10, not 7.

❺ The compilation rate is the number of lines compiled per minute of CPU time. CPU times are measured in 10-millisecond units.

Finally, when you use the /MACHINE_CODE option, a listing file is created showing the assembly language and machine code generated by the compiler. The default option to /MACHINE_CODE is INTERSPERSED; the other two options are BEFORE and AFTER. These options control the placement of the lines of machine code within the listing. Using the default option, the machine code is generated in line with the VAX C source statements and is listed below the last substituted line.

Example 1–5 shows the listing generated by the /MACHINE_CODE=BEFORE option.

Example 1–5: Machine Code Listing

```
EXAMPLE    14-JAN-1985 14:07:31    VAX C     V2.0-00  Page 1
V1.0       14-JAN-1985 14:02:10    DBA0:[.D]EXAMPLE.C;2  (1)


                          ❶    ❷
                          0000  main:

                     0000 0000     .entry  main,^m<>
               5E 10 C2   0002     subl2   #16,sp
   00000000* EF 16        0005     jsb     C$MAIN
5C 00000000* 8F D0        000B     movl    #SS$NORMAL,ap
               F2 AD 9F   0012     pushab  -14(fp)
00000000* EF 01 FB        0015     calls   #1,FTIME
               F2 AD DF   001C     pushal  -14(fp)
00000000* EF 01 FB        001F     calls   #1,CTIME
                  50 D5   0026     tstl    r0
                  OF 13   0028     beql    sym.1
                  50 DD   002A     pushl   r0
   00000000 EF DF         002C     pushal  $CHAR_STRING_CONSTANTS
00000000* EF 02 FB        0032     calls   #2,PRINTF

                          0039  sym.1:

00000000* EF 00 FB        0039     calls #0,PROCESS
               50 5C D0   0040     movl  ap,r0
                     04   0043     ret

    1           /*  This is a sample program to show the  *
    2            *  format of the compiler listing, as    *
    3            *  well as the effect of the various     *
    4            *  /SHOW command-line qualifier values   */
    5
    6           /*  This line shows what happens when a line
from the source file exceeds the right listing margin. */
    7
    8           #include "debugging.h"
   15
   16           #include timeb
   34
   35           #define NULL   0
   36
   37           main()
   38           {
   39     1        globalvalue  SS$NORMAL;
   40     1        struct       timeb  time_struct;
   41     1        char         *ctime_string, *ctime();
   42     1        int          status = SS$NORMAL;
   43     1
   44     1        ftime (& time_struct);
```

(Continued on next page)

Example 1–5 (Cont.): Machine Code Listing

```
   45    1          if ((ctime_string =
   46    1               ctime(& time_struct.time))
   47    1               != NULL)
   48    1
   49    1             printf ("Run time is %s",
   50    1                      ctime_string);
   51    1
   52    1       #if debugging
   53    X          printf ("\n*** Debugging version \
   54    X       ***\n\n");
   55    X
   56    X          status =
   57    1       #endif
   58    1
   59    1          process();
   60    1          return status;
   61    1       }


Command Line
------------

CC/LIST/MACHINE_CODE=BEFORE  EXAMPLE.C
```

Key to Example 1–5:

❶ The object module location of each statement and the machine code instructions are listed.

❷ The assembly language code generated by each line of source text is shown beside its corresponding machine code instruction.

Example 1–6 shows a listing file generated with the /DEFINE and /UNDEFINE command qualifiers.

Example 1–6: Listing Showing Command Line Definitions

```
EXAMPLE    14-JAN-1985 14:07:31    VAX C      V2.0-00  Page 1
V1.0       14-JAN-1985 14:02:10    DBA0:[.D]EXAMPLE.C;2  (1)


     1          /*  Compile this program with the    *
     2           *  following qualifiers...          *
     3           *                                   *
❶    4           *  /undefine=(vax)-                 *
     5           *  /define=(size=100,"func(n)")     */
     6
     7          #ifdef VAX
❷    8   X      #include stdio
     9          #else
    10          #include <stdio.h>
    68          #endif
    69
    70          main()
    71          {
    72    1        int i = SIZE;
❸          1      int i = 100;
    73    1        if (func(i))
❹          1      if (1)
    74    1           printf("%d\n",i);
    75    1      }


Command Line
------------

❺ CC/LIST/SHOW=(EXPAN,INTER)-
         EXAMPLE.C/UNDEFINE=(VAX)/DEFINE=(SIZE=100,func(n))
```

Key to Example 1–6:

❶ The command line qualifiers, as they were typed on the DIGITAL Command Language (DCL) command line, are different than the final DCL interpretation of the definitions.

❷ The identifier, VAX, is undefined, so the compiler ignores the first **#include** line and includes the STDIO.H file.

❸ The compiler replaces the defined identifier SIZE with defined value.

❹ The function func, declared on the DCL command line, returns a true value (1).

❺ This line shows how DCL translated the command line definition qualifiers: vax to VAX and size to SIZE. If defining or undefining from the command line, and you wish to specify an identifier containing lowercase letters, you must enclose the identifier in quotation marks as follows:

```
$ CC/LIST/SHOW=(EXPAN,INTER) -
_$ EXAMPLE/DEFINE=(size=100,"func(n)"))
```

For more information concerning the /DEFINE and /UNDEFINE qualifiers, refer to Section 1.3.2.

1.4 Linking a VAX C Program

Once you have compiled 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 VMS 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, you may want to use the /DEBUG qualifier when you 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 *VAX/VMS Linker Reference 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. Therefore, when 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 that of 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
```

Like the CC command, you can specify libraries on the LINK command line. The VAX C Run-Time Library functions are executed at run time, but references to these functions are resolved at link time. When you link your program, the linker resolves all references to VAX C Run-Time Library functions by searching any object code libraries that you 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$LIBRARY_n to the name of an object library and then searches that library for the code. You are to define the logicals LNK$LIBRARY_n as the libraries SYS$LIBRARY:VAXCCURSE, SYS$LIBRARY:VAXCRTLG, and SYS$LIBRARY:VAXCRTL. Depending on the needs of your program, you may have to access one, two, or all three libraries. In any case, you must adhere to the following rules for defining libraries for the linker to search.

1. If you do not need to use the Curses Screen Management package of VAX C RTL functions and macros, and you do not use the /G_FLOAT qualifier on the CC command line, you must define the logical as follows:

```
$ DEFINE  LNK$LIBRARY  SYS$LIBRARY:VAXCRTL [RETURN]
```

2. If you do plan to use the /G_FLOAT qualifier with the CC command line, but do not plan on using Curses, you must define the logicals as follows:

```
$ DEFINE  LNK$LIBRARY  SYS$LIBRARY:VAXCRTLG.OLB [RETURN]
$ DEFINE  LNK$LIBRARY_1  SYS$LIBRARY:VAXCRTL.OLB [RETURN]
```

3. If you do plan to use the Curses Screen Management package, but do not plan to use the /G_FLOAT qualifier, you must define the logicals as follows:

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

4. Finally, if you plan to use both Curses and the /G_FLOAT qualifier, you must define the three logicals in the following order:

```
$ DEFINE  LNK$LIBRARY  SYS$LIBRARY:VAXCCURSE.OLB [RETURN]
$ DEFINE  LNK$LIBRARY_1  SYS$LIBRARY:VAXCRTLG.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 possibly 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 concerning Curses, refer to the *VAX C Run-Time Library Reference Manual*. For more information concerning the G_floating representation of *double* variables, refer to Chapter 6, Data Types and Declarations.

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.

Using the object code of the VAX C RTL functions is one of two options. 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 program execution rate. For information concerning the shareable-image option, refer to the *VAX C Run-Time Library Reference Manual.*

1.4.2 LINK Command Qualifiers

The LINK command qualifiers can be used 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 *VAX/VMS Linker Reference Manual.*

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

/EXECUTABLE [=file-spec]
Causes the linker to produce an executable image. /NOEXECUTABLE suppresses production of an image file. The default is /EXECUTABLE.

/[NO]SHAREABLE
Causes the linker to create a shareable image. /NOSHAREABLE generates an executable image. The default is /NOSHAREABLE.

/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.

/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]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]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 object modules contained in an object module library with the /INCLUDE qualifier. The uses of object module libraries are described in Section 1.4.5.

- 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 *VAX/VMS Linker Reference Manual.*

The linker uses the following 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 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 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. 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 *VAX/VMS DCL Dictionary.*

For more information about object module libraries, see the *VAX/VMS Linker Reference Manual.*

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 attempt 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 you should verify that the modules will actually 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 Run-Time Library.

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

1.5 Running a VAX C Program

Once you have linked your program, you can use the DCL command RUN 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, Using the VMS Debugger.

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
The names of image modules that were 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
The name of the function in the calling sequence.

line
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.

relative PC
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.

absolute PC
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.

Chapter 2

Using the VMS Debugger

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

- An overview of the debugger
- Information to get you started using the debugger
- A sample terminal session that demonstrates using the debugger
- A list of the debugger commands by function

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

2.1 Overview

A debugger is a tool that helps you locate run-time errors quickly. It is used with a program that has already 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 other 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 issuing 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.

Such techniques allow you to isolate an error in your code much more quickly than you could without the debugger.

Once you have found the error in the program, you can then edit the source code and compile, link, and run the corrected version.

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

2.2 Features of the Debugger

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

- **Online HELP**

  Online HELP is always available during a debugging session and contains information on all the debugger commands and also information on 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. (Screen mode displays work best on VT100-series or VT200-series terminals or MicroVAX workstations.)

- **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 keypad).

- **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 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 issue 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

This section explains how to use the debugger with VAX C programs. The section focuses on basic debugger functions, to get you started quickly. It also provides any debugger information that is specific to VAX C. For more detailed information that is not specific to a particular language, see the *VAX/VMS Debugger Reference Manual.*

2.3.1 Compiling and Linking a Program to Prepare for Debugging

Before you can use 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 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).

You should 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, you should recompile it without the /NOOPTIMIZE qualifier.)

The /DEBUG qualifier on the LINK command 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

To invoke the debugger, issue the DCL command RUN. The following message will appear on your screen:

```
$ RUN INVENTORY

                    VAX DEBUG Version 4.n

%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 it 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, the following message will appear on your screen instead of the previous one when you invoke the debugger:

```
$ RUN INVENTORY

                    VAX DEBUG Version 4.n

%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).

The following message indicates that your program has completed successfully:

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

To interrupt a debugging session and return to DCL level, press CTRL/Y. This is useful if, for example, your program loops or you want to interrupt a debugger command that is still in progress.

To resume the debugging session after a CTRL/Y interruption, type either the CONTINUE or DEBUG command at DCL level. Use the CONTINUE command to return to the point at which you interrupted the debugging session. If you interrupted the session because of an infinite loop, use the DEBUG command instead. The DEBUG command returns you to the debugger prompt so that you can type another command. For example:

```
DBG> GO
     .
     .
     .
(infinite loop)
CTRL/Y
Interrupt

$ DEBUG
DBG>
```

To end a debugging session, type the EXIT command or press CTRL/Z:

```
DBG> EXIT
$
```

2.3.3 Issuing Debugger Commands

You can issue debugger commands any time you see the debugger prompt (DBG>). Type the command at the keyboard and press the RETURN key. You can issue 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 (-).

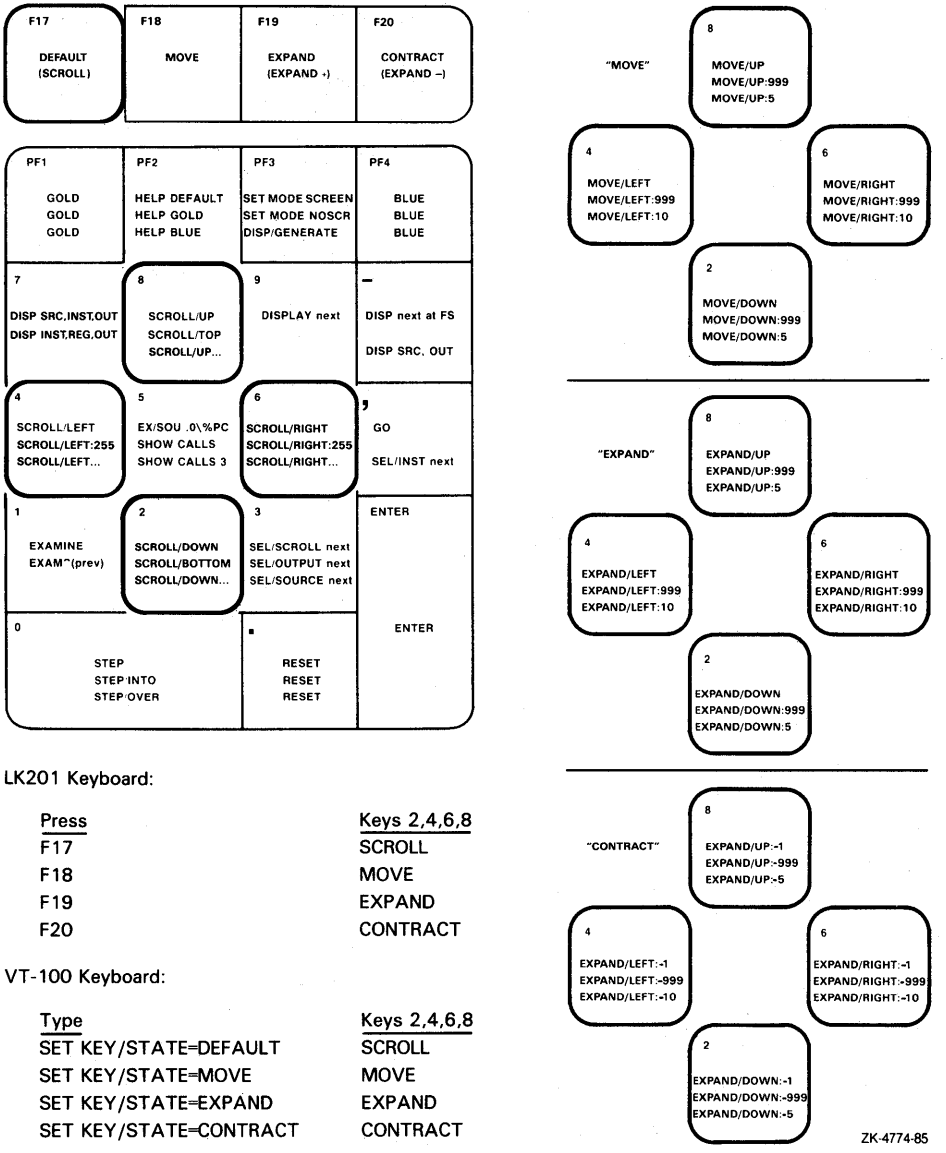
Alternatively, you can use the numeric keypad to issue certain commands. Figure 2–1 identifies the predefined key functions. You can also redefine key functions with the DEFINE/KEY command.

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 issues the STEP command; pressing key PF1 and then key 0 issues the STEP/INTO command; pressing key PF4 and then key 0 issues the STEP/OVER command.

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

Figure 2–1: Debugger Keypad Key Functions



LK201 Keyboard:

Press	Keys 2,4,6,8
F17	SCROLL
F18	MOVE
F19	EXPAND
F20	CONTRACT

VT-100 Keyboard:

Type	Keys 2,4,6,8
SET KEY/STATE=DEFAULT	SCROLL
SET KEY/STATE=MOVE	MOVE
SET KEY/STATE=EXPAND	EXPAND
SET KEY/STATE=CONTRACT	CONTRACT

2.3.4 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.

2.3.4.1 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
3:    i = 3;
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
```

2.3.4.2 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:  *           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) whose code is currently executing. 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 line numbers, which are assigned by the compiler, match those in a listing file.

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

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.

If the debugger cannot locate source lines for the routine that is currently executing, it tries to display source lines in the next routine down on the call stack for which source lines are available and issues the following message:

```
%DEBUG-I-SOURCESCOPE, Source lines not available for .0\%PC.
       Displaying source in a caller of the current routine.
```

Source lines may not be available for the following reasons:

- The PC value is within a system routine, or a shareable image routine for which no source code is available.

- The PC value is within a routine that was compiled without the /DEBUG compiler command qualifier (or with /NODEBUG).
- The PC value is within a routine whose module is not set (module setting is explained in Section 2.5.1).
- The source file was moved to a different directory after it was compiled (the location of source files is embedded in the object modules).

2.3.5 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 the current value of the program counter (PC) with the SHOW CALLS command
- Suspending program execution with breakpoints
- Tracing program execution with tracepoints
- Monitoring changes in variables with watchpoints

2.3.5.1 Starting and Resuming Program Execution

There are two 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 2.3.5.3, 2.3.5.4, and 2.3.5.5). If you set a breakpoint in the path of execution and then issue 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 issue commands. If you are using screen mode, the pointer in the source display indicates where execution stopped. You

can use the SHOW CALLS command (explained in Section 2.3.5.2) 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 the program by pressing CTRL/Y and then issue the DCL command DEBUG. You can then look at the source display and invoke a SHOW CALLS display to obtain the current PC value.

The STEP Command

The STEP command 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>
```

The PC value is now 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 issues 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.

Also, 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, although the routine is executed. Issuing the SET STEP INTO command causes the debugger to suspend execution within called routines, as well as within the routine that is currently executing.

2.3.5.2 Determining the Current Value of the Program Counter

The SHOW CALLS command lets you determine the current value of the program counter (PC) (for example, after returning to the debugger following a CTRL/Y interruption).

The SHOW CALLS command displays a traceback that lists the sequence of calls leading to the currently executing routine. For each routine (beginning with the currently executing routine), 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).

2.3.5.3 Suspending Program Execution

The SET BREAK command lets you select *breakpoints*, which are locations at which program execution is suspended. When you reach a breakpoint, you can issue 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 could step through the procedure COUNT, using the STEP command, and use the EXAMINE command (discussed in Section 2.3.6.1) 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.

Routine names and labels should be specified 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. (Otherwise, the debugger interprets the line number as a memory location.) For example, the next 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
```

Note that you can set breakpoints only on lines that resulted 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 always 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, issue the SHOW BREAK command:

```
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 issue 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 issuing the following commands:

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

To cancel a breakpoint, issue 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.

2.3.5.4 Tracing Program Execution

The SET TRACE command 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 issues 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, as well as 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
    .
    .
    .
```

2.3.5.5 Monitoring Changes in Variables

The SET WATCH command lets you set *watchpoints* that will be monitored continuously as your program executes. With high-level languages, you typically set watchpoints on variables that have been 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 issued 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 the following restriction when setting watchpoints. You can set watchpoints only on static variables. In VAX C, a static variable is created by specifying the **static**, **globaldef**, **globalref**, or **extern** storage class specifiers in the variable declaration.

You cannot set watchpoints on variables that are allocated on the stack or in registers. In VAX C, such nonstatic variables include **auto** and **register** variables.

If you try to set a watchpoint on a nonstatic variable, the debugger issues a message such as the following:

```
%DEBUG-W-BADWATCH, cannot watch protect address 7FF6E898
```

2.3.6 Examining and Manipulating Data

This section explains how to use the EXAMINE, DEPOSIT, and EVALUATE commands 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.

Note that, before you can examine or deposit into a nonstatic variable (as defined in the previous section), its defining routine must be active (on the call stack).

2.3.6.1 Displaying the Values of Variables

To display the current value of a variable, use the EXAMINE command. 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>
```

The EXAMINE command can be used 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 2.3.6.3 for a comparison of the EXAMINE and EVALUATE commands.

2.3.6.2 Changing the Values of Variables

To change the value of a variable, use the DEPOSIT command. 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 the data to be interpreted in some other data format.

2.3.6.3 Evaluating Expressions

To evaluate a language expression, use the EVALUATE command. 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 would adversely affect debugging. The VAX C operators that are supported in language expressions are listed in Table 2–1; VAX C operators that are not supported by the debugger are listed in Table 2–2.

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
|= &= ^=	
?:	Conditional
(type)	Cast

The following example shows how the EVALUATE and EXAMINE commands are similar. 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 of 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 the HELP LANGUAGE command at the DBG> prompt.)

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

2.4.1 Accessing Scalar Variables

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

Review Example 2–1, which presents the VAX C program SCALARS.C to be used in the next sample debugging session.

Example 2–1: Debugging Sample Program SCALARS.C

```
/*  SCALARS.C  This program defines a large number of          *
 *             variables so as 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 = 3.123456789012345678 9e10;
   ui = -438394;
   li = 790374270;
   ch = 'A';
   color = blue;
   ptr = &li;
}
```

The following debugging session executes SCALARS.EXE and illustrates 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 SHOW SYMBOL/TYPE command to display the data type of one variable.

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

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

The commands in the example set a breakpoint just prior to the end of the program and issue 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. Note that **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, the /ASCII qualifier is used to translate the byte integer into its ASCII equivalent.

2.4.2 Accessing Arrays

With the EXAMINE command, you can look at the values in arrays, using the usual 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. Keep in mind the differences between pointer arithmetic in VAX C and pointer arithmetic in other languages (for more information, refer to Chapter 6, Data Types and Declarations). Given 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]
```

Review Example 2–2, which presents the VAX C program ARRAY.C to be used in the next sample debugging session.

Example 2–2: Debugging Sample Program ARRAY.C

```
/*  ARRAY.C  This program increments an array so as 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 illustrates the commands used to access variable arrays:

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

The commands in the example 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 simply 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.

2.4.3 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, the /ASCIIZ qualifier (abbreviated as /AZ) can be used 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.

Review Example 2–3, which presents the VAX C program STRING.C to be used in the next sample debugging session.

Example 2–3: Debugging Sample Program STRING.C

```
/*  STRING.C  This program establishes a string so as 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 illustrates 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.

2.4.4 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.

References to members of a structure or union can be made 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, then 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 usual VAX C type-checking rules. 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 pathname with the reference to x to narrow the scope that is used and to resolve the ambiguity. The same pathname is used to look up both x and y.

Review Example 2–4, which presents the VAX C program STRUCT.C to be used in the next sample debugging session.

Example 2–4: Debugging Sample Program STRUCT.C

```
/*  STRUCT.C  This program defines a structure and union so as *
 *            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';

}
```

The following debugging session executes STRUCT.EXE and illustrates 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 command in this sample debugging session is as follows:

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

Setting a breakpoint at line 29 and issuing 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 in *p and p −−> bf in the previous example.

The next command in this sample debugging session is 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. Therefore, 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.

Review Example 2–5, which presents the VAX C program ARSTRUCT.C to be used 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 illustrates 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 breakpoint conditionally.

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 the usual VAX C syntax for referring 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 probably will 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 which uses the VAX C syntax for arithmetic expressions.

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

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

The debugger issues 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

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 surrounding the current PC value. 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 issues a warning. For example:

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

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

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

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

Note that dynamic module setting may slow down the debugger as more and more modules are set. If performance becomes a problem, you can use the CANCEL MODULE command to reduce the number of set modules, or you can disable dynamic module setting by issuing the SET MODE NODYNAMIC command. (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 scope and visibility rules of the currently set language. In general, the debugger first searches for a symbol within the block or routine surrounding the current PC value. If the symbol is not found in that scope region, the debugger searches the nesting program unit, then its nesting unit, and so on. (The precise order of search depends on the currently set 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 at the current PC value, the debugger also searches other scope regions. First, it searches within the currently executing routine, then the caller of that routine, then its caller, and so on, until the symbol is found. Symbolically, this search list is denoted 0,1,2, . . . ,*n*, where *n* is the number of calls in the call stack. Within each of these scope regions, the debugger uses the visibility rules of the currently set language to locate symbols.

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

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

You can then 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. Then, references to Y without a path-name prefix will 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 CANCEL SCOPE command.

2.6 Sample Debugging Session

Example 2–6 shows the VAX C program POWER.C, which is to be used in a simple debugging session. To learn about a larger number of debugger commands, reexecute this program and use some of the commands listed in Section 2.7.

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 illustrates some of the debugger commands used to evaluate the execution of POWER.C.

Example 2–7: A Sample Debugging Session

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

       VAX DEBUG Version 4.n

❷ %DEBUG-I-INITIAL, language is C, module set to 'POWER'
❸ DBG> set break  %LINE 13
❹ DBG> go
❺ break at POWER\main\%LINE 13
❻     13:    j = power(i);
❼ DBG> step/into
❽ stepped to routine POWER\power
      16: int j;
  DBG> step
  stepped to POWER\power\%LINE 18
      18:    return (j * j)
❾ DBG> examine J
❿ %DEBUG-W-NOSYMBOL, symbol 'J' is not in the symbol table
  DBG> examine j
⓫ 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
⓬ POWER\main\j:  4
  DBG> go
⓭ %DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful
                           completion'
⓮ DBG> exit
  $
```

The following numbers correspond to the numbers in the previous example:

❶ 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. You should use the /NOOPTIMIZE qualifier to turn off compiler optimization that may interfere with debugging. If you desire a

minimal amount of optimization that would not interfere with your debugging session, use the /OPTIMIZE=NODISJOINT qualifier.

❷ 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.

❸ You enter debugger commands at the 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.

❹ The debugger command GO begins execution of the image.

❺ 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 in which object module it is working, 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.

❻ 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.

❼ 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 /INTO. Use STEP/INTO to step through a user-defined or VAX C Run-Time Library function.

❽ 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 (\).

❾ The debugger command EXAMINE displays the contents of a variable.

❿ The debugger does not recognize the variable, J, as existing in the scope of the current module.

⓫ The debugger supports the case sensitivity of VAX C variables; variable j exists whereas 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.

⓬ The value of variable j in function main is different than the separate variable j in function power. Function power executes properly; it returns the number 2 raised to the second power.

⓭ Upon completion of execution of the image, the debugger states the status of the execution. In this example, execution was successful.

⓮ To issue 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.

The following section explains some of the commonly used debugger commands.

2.7 Debugger Command Summary

This section lists all of the debugger commands and any related DCL commands in functional groupings, along with brief descriptions.

During a debugging session, you can get online HELP on any command and its qualifiers by typing the HELP command followed by the name of the command in question. The HELP command has the following form:

```
HELP command
```

2.7.1 Starting and Terminating a Debugging Session

($) RUN[1]	Invokes the debugger if LINK/DEBUG was used.
($) RUN/[NO]DEBUG[1]	Controls whether the debugger is invoked when the program is executed.

[1] This is a DCL command, not a debugger command.

CTRL/Z or EXIT	Ends a debugging session, executing all exit handlers.
QUIT	Ends a debugging session without executing any exit handlers declared in the program.
CTRL/Y	Interrupts a debugging session and returns you to DCL level.
CTRL/C	Has the same effect as CTRL/Y, unless the program has a CTRL/C service routine.
($) CONTINUE[1]	Resumes a debugging session after a CTRL/Y interruption.
($) DEBUG[1]	Resumes a debugging session after a CTRL/Y interruption but returns you to the debugger prompt.
ATTACH	Passes control of your terminal from the current process to another process (similar to the DCL command ATTACH).
SPAWN	Creates a subprocess; lets you issue DCL commands without interrupting your debugging context (similar to the DCL command SPAWN).

[1] This is a DCL command, not a debugger command.

2.7.2 Controlling and Monitoring Program Execution

GO	Starts or resumes program execution.
STEP	Executes the program up to the next line, instruction, or specified instruction.
{ SET | SHOW } STEP	Establishes or displays the default qualifiers for the STEP command.
{ SET | SHOW | CANCEL } BREAK	Sets, displays, or cancels breakpoints.
{ SET | SHOW | CANCEL } TRACE	Sets, displays, or cancels tracepoints.

{ SET, SHOW, CANCEL } WATCH	Sets, displays, or cancels watchpoints.
{ SET, CANCEL } EXCEPTION BREAK	Sets or cancels exception breakpoints.
SHOW CALLS	Identifies the currently active routine calls.
SHOW STACK	Gives additional information about the currently active routine calls.
CALL	Calls a routine.

2.7.3 Examining and Manipulating Data

EXAMINE	Displays the value of a variable or the contents of a program location.
DEPOSIT	Changes the value of a variable or the contents of a program location.
EVALUATE	Evaluates a language or address expression.

2.7.4 Controlling Type Selection and Symbolization

{ SET, SHOW, CANCEL } RADIX	Establishes the radix for data entry and display, displays the radix, or restores the radix.
{ SET, SHOW, CANCEL } TYPE	Establishes the type to be associated with untyped program locations, displays the type, or restores the type.
SET MODE [NO]G_FLOAT	Controls whether double-precision floating-point constants are interpreted as G_FLOAT or D_FLOAT.
SET MODE [NO]LINE	Controls whether code locations are displayed in terms of line numbers or routine-name + byte offset.

SET MODE [NO]SYMBOLIC	Controls whether code locations are displayed symbolically or in terms of numeric addresses.
SYMBOLIZE	Converts a virtual address to a symbolic address.

2.7.5 Controlling Symbol Lookup

SHOW SYMBOL	Displays symbols in your program.
{ SET / SHOW / CANCEL } MODULE	Sets a module by loading its symbol records into the debugger's symbol table, identifies a set module, or cancels a set module.
{ SET / SHOW / CANCEL } IMAGE	Sets a shareable image by loading data structures into the debugger's symbol table, identifies a set image, or cancels a set image.
SET MODE [NO]DYNAMIC	Controls whether modules and shareable images are set automatically when the debugger interrupts execution.
{ SET / SHOW / CANCEL } SCOPE	Establishes, displays, or restores the scope for symbol lookup.

2.7.6 Displaying Source Code

TYPE	Displays lines of source code.
EXAMINE/SOURCE	Displays the source code at the location specified by the address expression.
{ SET / SHOW / CANCEL } SOURCE	Creates, displays, or cancels a source directory search list.
SEARCH	Searches the source code for the specified string.
{ SET / SHOW } SEARCH	Establishes or displays the default qualifiers for the SEARCH command.

{ SET / SHOW } MAX_SOURCE _FILES	Establishes or displays the maximum number of source files that may be kept open at one time.
{ SET / SHOW } MARGINS	Establishes or displays the left and right margin settings for displaying source code.

2.7.7 Using Screen Mode

SET MODE [NO]SCREEN	Enables or disables screen mode.
SET MODE [NO]SCROLL	Controls whether an output display is updated line by line or once per command.
DISPLAY	Modifies an existing display.
{ SET / SHOW / CANCEL } DISPLAY	Creates, identifies, or deletes a display.
{ SET / SHOW / CANCEL } WINDOW	Creates, identifies, or deletes a window definition.
SELECT	Selects a display for a display attribute.
SHOW SELECT	Identifies the displays selected for each of the display attributes.
SCROLL	Scrolls a display.
SAVE	Saves the current contents of a display and writes it to another display.
EXTRACT	Saves a display or the current screen state and writes it to a file.
EXPAND	Expands or contracts a display.
MOVE	Moves a display across the screen.
{ SET / SHOW } TERMINAL	Establishes or displays the height and width of the screen.
CTRL/W or DISPLAY/REFRESH	Refreshes the screen.

2.7.8 Editing Source Code

EDIT	Invokes an editor during a debugging session.
{ SET / SHOW } EDITOR	Establishes or identifies the editor invoked by the EDIT command.

2.7.9 Defining Symbols

DEFINE	Defines a symbol as an address, command, or value.
DELETE or UNDEFINE	Deletes symbol definitions.
{ SET / SHOW } DEFINE	Establishes or displays the default qualifier for the DEFINE command.
SHOW SYMBOL/DEFINED	Identifies symbols that have been defined.

2.7.10 Using Keypad Mode

SET MODE [NO]KEYPAD	Enables or disables keypad mode.
DEFINE/KEY	Creates key definitions.
DELETE/KEY or UNDEFINE/KEY	Deletes key definitions.
SET KEY	Establishes the key definition state.
SHOW KEY	Displays key definitions.

2.7.11 Using Command Procedures and Log Files

DECLARE	Defines parameters to be passed to command procedures.
{ SET / SHOW } LOG	Specifies or identifies the debugger log file.
SET OUTPUT [NO]LOG	Controls whether a debugging session is logged.

SET OUTPUT [NO]SCREEN_LOG	Controls whether, in screen mode, the screen contents are logged as the screen is updated.
SET OUTPUT [NO]VERIFY	Controls whether debugger commands are displayed as a command procedure is executed.
SHOW OUTPUT	Displays the current output options established by the SET OUTPUT command.
{ SET / SHOW } ATSIGN	Establishes or displays the default file specification that the debugger uses to search for command procedures.
@file-spec	Executes a command procedure.

2.7.12 Using Control Structures

IF	Executes a list of commands conditionally.
FOR	Executes a list of commands repetitively.
REPEAT	Executes a list of commands repetitively.
WHILE	Executes a list of commands conditionally.
EXITLOOP	Exits an enclosing WHILE, REPEAT, or FOR loop.

2.7.13 Additional Commands

SET PROMPT	Specifies the debugger prompt.
SET OUTPUT [NO]TERMINAL	Controls whether debugger output is displayed or suppressed, except for diagnostic messages.
{ SET / SHOW } LANGUAGE	Establishes or displays the current language.
{ SET / SHOW } EVENT_FACILITY	Establishes or identifies the current run-time facility for language-specific events.
SHOW EXIT_HANDLERS	Identifies the exit handlers declared in the program.

{ SET / SHOW } TASK	Modifies the tasking environment or displays task information.
{ DISABLE / ENABLE / SHOW } AST	Disables the delivery of ASTs in the program, enables the delivery of ASTs, or identifies whether delivery is enabled or disabled.

VAX C Programming Concepts

Chapter 3

Program Structure

This chapter introduces the basic features of VAX C to the experienced programmer. The text provides detailed examples and short tutorials, as well as many pointers to other chapters in this manual.

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 VAX C function definitions, function declarations, and the following components of program structure:

- Function definitions
- Function declarations
- Function prototypes
- Function parameters and arguments
- Program identifiers
- Blocks
- Comments
- VAX C language keywords
- LINT-like functionality

3.1 C Programming Language Background

The C language is a general-purpose programming language that is manageable because of its small size, flexible because of its ample supply of operators, and powerful in its utilization of modern control flow and data structures. The C language was originally designed and implemented on a UNIX® system on the PDP-11. The designer 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 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. The C language manipulates and stores data with regard to the similarities of modern machine architecture. However, C is not as complex as assembler language and is not machine dependent. C is highly portable (a program is portable if you can compile and run its source program using several different compilers on several different machines).

Although there is no ANSI or other industry-wide standard for the C programming language, there is a consistency of functionality between implementations. There must be such 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.

Since the C language was developed in a UNIX system environment, and eventually was used to rewrite most of that operating system, 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.

® UNIX is a registered trademark of American Telephone and Telegraph Company.

1 Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Englewood Cliffs, New Jersey: Prentice-Hall, 1978), p. 1.

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 any action is taken by the compiler.

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.

3.2 The VAX C Programming Language

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, the VAX C user, must make a choice as to which set of features of VAX C are most important to your programming needs: portability across systems or highly 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. In order for the VAX C compiler 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, whereas a UNIX system accesses a file using a structure called a file descriptor, VAX Record Management Services (RMS) access files using a variety of control structures. In VAX C, input/output functions appear to access files in the same manner as UNIX systems, but the compiler 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 the user, but can slow the execution of a program in some instances.

Most of the differences between VAX C and other implementations—differences that hinder portability of source code—evolved due to the differences between VMS and UNIX. For example, it is difficult for VAX C to create an environment that allows the programmer a great amount of control, as in a UNIX system environment, when VMS will not grant a user such control; I/O redirection is not a part of the VMS command line syntax; creating subprocesses in VMS is not as efficient as it is in UNIX; and, VMS high-level languages do not implement preprocessors in exactly the same manner as languages on a UNIX system. In this manual, 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 VMS 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 routines within VAX C programs.

However, you can have portability and efficiently access the powerful VMS environment also. You can use special constructs of VAX C and of 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 VMS, and to execute other segments of code when running on systems other than VMS. For a complete discussion of portability, refer to the *VAX C Run-Time Library Reference Manual.* For more information concerning the preprocessor directives, refer to Chapter 8, Preprocessor Directives. For information concerning the DCL command line, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

3.3 Writing a Program

Although conceptually simple, writing the first program in an unfamiliar language can be a frustrating experience. Since computers are known for their efficient processing of numbers, the first program presented here is one which adds two numbers and stores the total in a variable. Example 3–1 shows how to code such a program.

Example 3–1: 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   */
```

The following numbers correspond to the numbers in the previous example:

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 appears. *White space* is an area within the source code where blank spaces or blank lines separate code. In following chapters, allowable 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 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 concerning the syntax and usage of the main_program option, refer to Section 3.8.1.

VAX C functions have methods of exchanging information using parameters and arguments. In the function definition of main, there are no parameters designated by the empty parentheses; therefore, in the previous example, the function main cannot receive information by use of parameters.

In order to specify parameters in a function definition, you 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 the previous example, 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 analogous to the **DO-END** of PL/I, or the **BEGIN-END** of Pascal. Usually, the body contains one or more **return** statements. A **return** statement specifies what, if anything, is returned to the expression that called the function. Depending upon the set-up of the function, you can omit the **return** statement, and its return value would remain undefined. If a function does not return a value, you

can declare the function to be of data type **void**. For more information concerning functions, refer to Section 3.8.2.

❸ In the example, the variable total is declared and defined within the function main. You must 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 of type **int** 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. The user 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 concerning data types, refer to Chapter 6, Data Types and Declarations. For more information concerning storage classes, refer to Chapter 7, Storage Classes and Allocation.

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**) are called *keywords*. Keywords are identifiers that are predefined identifiers and cannot be redeclared. You cannot use these words to identify variables and functions in your programs. Keywords must be expressed in lowercase letters. For a list of the VAX C keywords, refer to Section 3.13.

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. You do not have to be as specific when referencing all VAX C identifiers; all instances of case sensitivity are flagged in this manual.

❹ Finally, 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 concerning the various VAX C statements, refer to Chapter 4, Statements. For more information concerning the VAX C operators, refer to Chapter 5, Expressions and Operators.

3.4 Producing Input/Output

The C language includes no facilities to administer input and output (I/O). However, all implementations must have methods by which the programs and users communicate. The lack of communication in the previous example 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 Run-Time Library 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.

VAX C RTL functions are segments of object code (and, as an option, shareable images) that are accessed when external references within your program are resolved. All the RTL object code modules are located in the libraries SYS$LIBRARY:VAXCCURSE.OLB, SYS$LIBRARY:VAXCRTLG.OLB, and SYS$LIBRARY:VAXCRTL.OLB. Before you can execute any of the example programs in this manual, you *must* define, in the correct order, the libraries the linker must search to resolve references to VAX C RTL functions. To determine in which order to define these libraries, refer to the *VAX C Run-Time Library Reference Manual*. For general information concerning libraries, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

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 RETURN
```

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 also can locate the .H definition files in text library VAXCDEF.TLB located in directory SYS$LIBRARY. This manual refers to the .H files as definition modules since they can be accessed as modules in this text library.

For more information concerning macros, refer to Chapter 8, Preprocessor Directives. For more information on the various methods of accessing VAX C RTL functions, refer to the *VAX C Run-Time Library Reference Manual.*

Example 3–2 shows that by using the VAX C RTL function **printf**, a VAX C program can print a message to the terminal.

Example 3–2: Output of Information

```
/*  This program adds two numbers, assigns the value 4 to *
 *  variable total, and then prints the answer on the     *
 *  terminal screen.                                      */

main()
{
   int total;

   total = 2 + 2;
                              /* Print intro string       */
❶  printf("Here is the answer: ");
   printf("%-d.", total);     /* Print the answer         */
}
```

The following number corresponds to the number in the previous example:

❶ The VAX C RTL function **printf** writes to the standard output (the terminal screen). The first call to the 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 letter d forces the value of the argument to be expressed as a decimal number. The period (.) prints immediately after the value of total.

Output from this program 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 3–3 illustrates how to output on two lines.

Example 3–3: 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.                     */

main()
{
   int total;

   total = 2 + 2;
                                   /* Print intro string       */
   printf("Here is the answer...\n");
                                   /* Print the answer         */
   printf("%-d.", total);
}
```

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 DIGITAL Command Language (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 by a file extension. You should choose the file name so that the file is easily identifiable to the user. You should 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.

Once your program has been created and named, the program can be compiled, linked, and executed as follows:

```
$ DEFINE  LNK$LIBRARY  SYS$LIBRARY:VAXCRTL.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 in order to use VAX C RTL functions in your program. The definition in the previous example is sufficient to execute all example programs in this chapter. Once you have defined the libraries, you do not have to define them again for the remainder of the terminal session (until you log out). For more information concerning the specification of libraries to the linker, the creation of source code, and the compilation process, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

3.5 Controlling Program Flow

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 concerning the **while** statement, another statement which loops until meeting a condition, refer to Chapter 4, Statements.

3.5.1 The if Statement

When executing one or more VAX C statements given a certain condition, you can use the **if** statement. Example 3–4 shows a program using the **if** statement.

Example 3–4: 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.                             */

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')
                               /* If correct guess         */
      printf("You're right!");
   else                        /* If incorrect guess       */
      {
         printf("You're wrong.\n");
         printf("You'll have to try again!");
      }
}
```

The following numbers correspond to the numbers in the previous example:

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.

❷ If the letter that the user types is either 'a' or 'A', then a message stating that the choice is correct prints. 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 concerning blocks, refer to Section 3.14.

Sample output from this program is as follows:

```
$ RUN  EXAMPLE4 RETURN
Guess which letter I'm thinking of!
B RETURN
You're wrong.
You'll have to try again!
```

3.5.2 The switch Statement

The **switch** statement can perform the same task as the **if** statement does in the previous example, but **switch** is particularly useful when many conditions must be tested. An example of the **switch** statement is as follows:

Example 3–5: Conditional Execution Using the switch Statement

```
/*  This program plays the same guessing game as the        *
 *  previous example except that it uses the switch         *
 *  statement.                                              */

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!");
       }
}
```

The following numbers correspond to the numbers in the previous example:

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 SYS$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 which contains macros and definitions used by the RTL functions; there are several ways to include definitions in the program stream. For more information concerning the VAX C RTL and the definition modules, refer to the *VAX C Run-Time Library Reference Manual.*

❷ 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. For more information concerning the possible side effects of macros, refer to Chapter 8, Preprocessor Directives.

Output from this program 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 the previous example, the **_tolower** macro translated all answers to lowercase letters, so there is no need to test for uppercase letter 'A'. When a case label has been matched with the value of expression ch, all of the statements following all of 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 concerning **switch** statements, refer to Chapter 4, Statements.

3.5.3 Loops

In the previous examples, the user 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, thereby guaranteeing at least one execution of the body; in VAX C, this last loop is called the **do** statement. Example 3–6 shows that you can use the **do** statement to alter the letter-guessing program.

Example 3–6: 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 ctype

main()
{
   char  ch;

   printf("Guess what letter I'm thinking of!\n");
   printf("Keep guessing till you get it!\n");

   do                         /* Do the following ...       */
      {                       /* Beginning of loop body     */
         ch = getchar();
         ch = _tolower(ch);
         switch(ch)
            {
               case 'a' :
                  printf("You're right!");
                  return;

                              /* Ignore RETURN (newline) ch */
❶              case '\n':
                  break;

               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     */
❷  while(ch != 'a' && ch != 'A');
}
```

The following numbers correspond to the numbers in the previous example:

❶ 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.

❷ In the **while** expression at the end of the **do** statement, the logical AND operator (&&) presents the condition to be tested. The **while** expression 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'."

Output from this program 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!
```

The **for** statement can be used 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. You can use other loops to limit the amount of guesses, but you are responsible for incrementing a counter, whereas the **for** statement increments automatically. Example 3–7 illustrates the use of the **for** statement.

Example 3–7: 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 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!");
}
```

The following numbers correspond to the numbers in the previous example:

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 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 each time the value of variable i increases by 1, until the value of i is greater than 3.

❷ The double minus signs (--) are the decrement operator. The decrement operator is used in this example to subtract one from the value of variable i so that newline characters would not be counted as the guess of a letter.

Sample output from the program 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!
```

3.6 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:

```
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 many things, such as integers, real numbers, character strings, or data structures. One type of integer that it can be is 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  *pointr;
```

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:

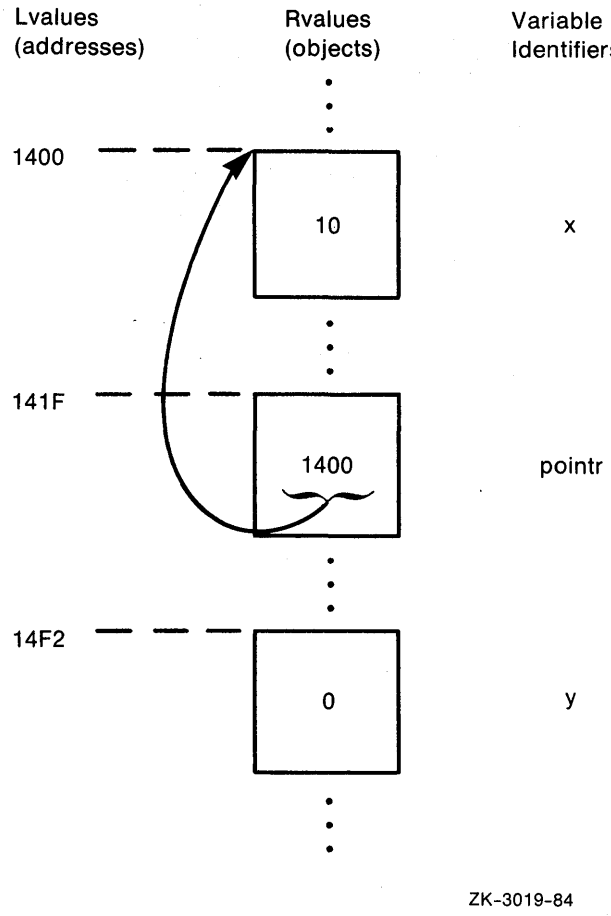
```
static  int *pointr;              /* Declarations              */
static  int x = 10, y = 0;

        pointr = &x;              /* Assignment                */
```

The rvalue of the variable pointr is the lvalue of variable x. Notice that 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 keyword **static** specifies the **static** storage class. For information concerning **static** and other storage class specifiers and modifiers, refer to Chapter 7, Storage Classes and Allocation.

Figure 3–1 illustrates the difference between rvalues and lvalues.

Figure 3–1: rvalues, lvalues, and Assigning Pointers



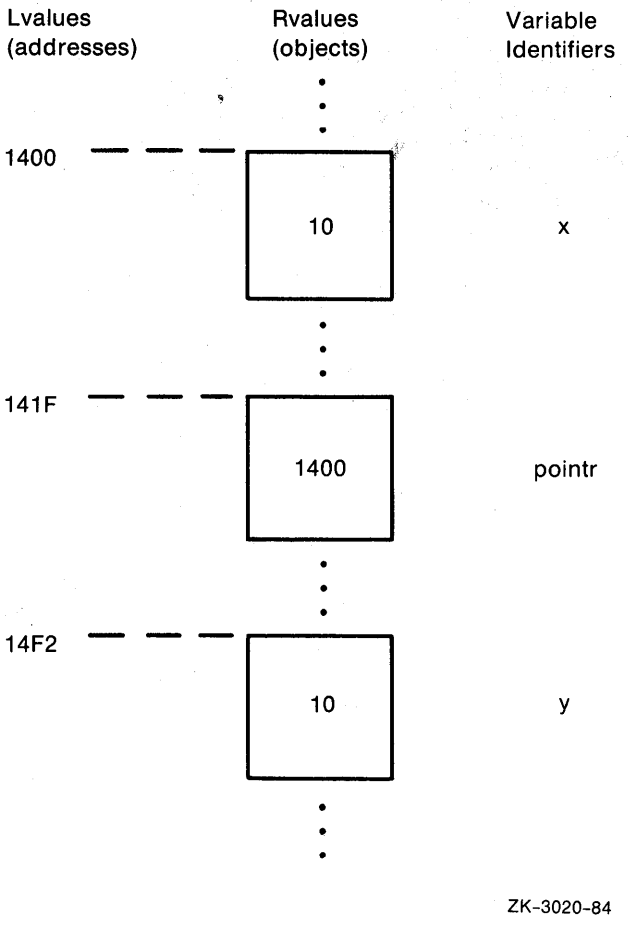
Notice how the value of the variable pointr contains the address of variable x. Remember that the location of variables in memory and the order in which the compiler processes them is unpredictable and left to the discretion of the compiler.

Once you have assigned an address to the pointer, you will want to use it. For example, if you wanted to assign x's rvalue to a variable y, you could use the pointer in a VAX C statement as follows:

```
y = *pointr;
```

The asterisk (*) is the VAX C indirection operator; the object of the variable being pointed to by pointr 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 3–2 shows the status of the variables after you execute the last code example.

Figure 3–2: The Indirection Operator in Assignments



For more information concerning pointers, refer to Chapter 6, Data Types and Declarations.

3.7 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 which you can declare to be of a scalar or an aggregate data type.

3.7.1 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**. A character string may be declared and initialized 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 strng[] =   "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 program, you must use the string-handling VAX C Run-Time Library functions. For more information concerning the string-handling functions, refer to the *VAX C Run-Time Library Reference Manual*. Example 3–8 illustrates the use of character strings and arrays.

Example 3–8: 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.                               */

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);
      }
}
```

Output from the program is as follows:

```
$ RUN  EXAMPLE8 [RETURN]
Guess which letter I'm thinking of!
B [RETURN]
You're wrong.
You'll have to try again!
```

For more information concerning arrays and character strings, refer to Chapter 6, Data Types and Declarations.

3.7.2 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;
}
```

Members may be referenced 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 3–9 illustrates the nature of unions.

Example 3–9: Single Storage Allocation of Unions

```
/*  This example illustrates the storage maintenance of        *
 *  unions with different size members.                        */

main()
{
   union                            /* Declare the 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);
}
```

Output from this example is as follows:

```
$ RUN  EXAMPLE9.EXE [RETURN]
Jackson
M
Mackson
```

The RTL function **strcpy** copies the second string into the array variable. 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 concerning structures and unions, refer to Chapter 6, Data Types and Declarations.

Example 3–10 shows a structure definition and its usage.

Example 3–10: Structures

```
/*  This program plays the same guessing game as the      *
 *  previous examples except that it uses a structure.    */

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        */
❶  struct  storage            /* Structure tag = storage  */
      {
         char       small_a;  /* One correct letter       */
         char     capital_a;  /* Another correct letter   */
         char    newline_ch;  /* newline character        */
         int    num_guesses;  /* Number of guesses        */
      } the_var;              /* Dummy declaration        */

                              /*  Declare "letter"        *
                               *   using tag "storage"    */
❷  struct  storage  letter = {'a', 'A', '\n'};

   letter.num_guesses = 3;
   printf("%s\n", greeting1);
   printf("%s\n", greeting2);
```

(Continued on next page)

Example 3–10 (Cont.): Structures

```
    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);
                 }

       }                    /* End of for loop body       */
       printf("%s", message4);
}
```

The following numbers correspond to the numbers in the previous example:

❶ 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**. The dummy variable the_var initiates storage allocation and enables the tag storage to be used in structure declarations elsewhere in the program.

❷ 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.

Output from the program 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.

3.8 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 is dependent on what type of value the function returns, and the position of the function definition within the program. This section explains the rules for defining functions, but you may wish to refer to the discussion of declarations and definitions in Chapter 6, Data Types and Declarations, before reading further.

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 3.11 for a detailed discussion of parameters and arguments.

Example 3–11 presents an example of two function definitions.

Example 3–11: 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()
❶ {
   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                   */
❷ lower(c_up)
❸ 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  */
```

The following numbers correspond to the numbers in the previous example:

❶ 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. For more information concerning the **return** statement, refer to Chapter 4, Statements.

❷ 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.

❸ 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 in 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 concerning data types, refer to Chapter 6, Data Types and Declarations.

For more information concerning the VAX C operators used in the previous example, refer to Chapter 5, Expressions and Operators.

3.8.1 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 3–11, the main function physically precedes the function lower, but the two function definitions could 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 slightly 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 concerning scope and storage classes, refer to Chapter 7, Storage Classes and Allocation.

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 function 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.

3.8.2 Parameter List Declarations

Example 3–11 illustrates only one of two possible methods of listing function parameters, as follows:

```
lower( c_up )
int  c_up;
{
    .
    .
    .
```

To make your code concise, you may wish to 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 3.10 for more information concerning 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:

1. You can specify empty parentheses.
2. You can use the keyword **void** to specify an empty parameter list.

An example of the use of the **void** keyword is as follows:

```
char function_name( void )
{ return 'a'; }
```

3.8.3 Function Return Data Types

By default, all VAX C functions return objects of data type **int**. In Example 3–11, 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 illustrates 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 which requires a value, you can define the function to return a value of type **void**. Use of the **void** keyword 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 **void** keyword with the cast operator for anything other than a function.

The following example illustrates the use of 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;
}
```

3.8.4 Variable-Length Parameter Lists

If you decide to define a function with a variable-length parameter list, you can use ellipses (. . .) to designate the variable-length portion of the parameter list, as follows:

```
function_name( int lower, int upper, int temp, char x, float y, ... )
{
   .
   .
   .
```

Within the function body, you should use the varargs functions and macros to access the argument list passed to the function. The varargs functions and macros provide a portable means of accessing variable-length argument lists. For more information concerning variable-length argument lists, refer to the varargs 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 varargs header and declaration within the parameter list and before the function body, as opposed to using the ellipsis notation. The following example illustrates such a construct:

```
function_name( lower, upper, temp, x, y, va_alist )
int lower, upper, temp;
char x;
float y;
va_dcl
{
    .
    .
    .
```

NOTE

If you use function prototypes, you should use ellipses (. . .) within parameter lists so that the compiler does not compare varargs declarations (va_alist, va_dcl) with prototype data declarations. See Section 3.10 for more information concerning function prototypes.

3.9 Function Declarations

As in Example 3–11, 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 3–12 illustrates when you need to declare a function.

Example 3–12: 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);
      }
}

char lower(c_up)           /* The function definition  */
int c_up;
{
    .
    .
    .
}
```

The following number corresponds to the number in the previous example:

❶ 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, despite lower's return data type, you would not have to declare function lower before you call the function.

In a function declaration, you can use the keyword **void** to specify an empty argument list, as follows:

```
main()
{
   char function_name( void );
      .
      .
      .
}
char function_name( void )
{ }
```

If the function's return data type is **void**, you must use the keyword in the declaration, as follows:

```
main()
{
   void function_name( void );
      .
      .
      .
}
void function_name( void )
{ }
```

If you specify argument data types or **void** in 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 3.10 for more information concerning function prototypes.

3.10 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 3.16 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 )
{ }
```

Or, equivalently:

```
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 floating point value. The type specifications are identical to the ones used in a parameter list located before the function body. For more information concerning the interpretation of complex declarations, refer to Chapter 6, Data Types and Declarations.

In each compilation unit in your program, you should determine the position in which 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 of 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 actually 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 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 of 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 ellipses. The following example illustrates 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 appropriate error messages:

```
char  function_name( lower, *upper, char (*func)(), float y );
char  function_name( , , char (*func)(), float y );
char  function_name( ... );
```

3.10.1 Using Function Prototypes

The use of the 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 has been 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 3.11).

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.

Also, if you wish to use prototype checking for VAX C Run-Time Library (RTL) function calls, you can include the module or modules appropriate for the RTL functions used in your program. You place the include preprocessor directives at the top of any applicable compilation units.

For more information concerning the RTL prototype include modules, refer to the *VAX C Run-Time Library Reference Manual.* For more information concerning preprocessor directives, refer to Chapter 8, Preprocessor Directives. For more information concerning compilation units and scope, refer to Chapter 7, Storage Classes and Allocation.

3.11 Using Parameters and Arguments

VAX C functions can exchange information by means of parameters and arguments. (In this manual, 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 3–11, 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 always 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. You should not expect function calls or other complicated expressions in the argument list to be evaluated in any particular order.
- In VAX C, all 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. The rule applies to all scalar variables, structures, and unions passed as arguments. 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 Calling Standard. For more information concerning the VAX Calling Standard and passing arguments in VAX C, refer to Chapter 10, Mixed-Language Programming.

- 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 and thus does not generally convert the arguments to the types of the parameters. Instead, all of 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 types **char** or **short** are converted to **int**.
  - Any arguments of types **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. The corresponding parameter must be declared 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. The corresponding parameter can be declared 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 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 concerning the cast operator, refer to Chapter 5, Expressions and Operators.
- 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**.

3.11.1 Function and Array Identifiers as Arguments

A function identifier can also be used 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 3–13 shows when you must declare user-defined functions and how to pass functions as arguments.

Example 3–13: Declaring Functions Passed as Arguments

```
                                /* Defined before it is     *
                                 *   used                   */
❶ x()  { return 25; }

  main()
  {
❷    int y();                   /* Function declaration     */
        .
        .
        .
❸    funct(x, y);               /* Passed as addresses      */
        .
        .
        .
  }

  y() { return 30; }            /* Function definition      */


  funct(f1, f2)                 /* Function definition      */
                                /* Declare arguments as     *
                                 *  pointers to functions   *
                                 *  returning an integer    */
❹ int (*f1)(), (*f2)();
  {
     (*f1)();                   /* A call to a function     */
        .
        .
        .
  }
```

The following numbers correspond to the numbers in the previous example:

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.

❹ When declaring the function identifiers as parameters, declare the function as the result of the indirection operator (*) applied to the address of the function. For more information concerning parentheses in expressions and the indirection operator, refer to Chapter 5, Expressions and Operators.

VAX C treats array parameters in the same way. If you pass an array identifier in an argument list, VAX C translates the identifier as a pointer to the data type of the array elements. In order to access the first element of the array, you need to dereference the pointer. For more information concerning pointers, addresses, and dereferencing, refer to Chapter 6, Data Types and Declarations.

3.11.2 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[ ];
```

In this syntax, parameter argc is the count of arguments present in the command line that invoked the program, and parameter argv is a character-string array of the arguments. Parameter envp is the environment array. It 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. (For more information, refer to the *VAX C Run-Time Library Reference Manual*).

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 always greater than or equal to 1, and argv[0] always contains the name of the image file.

Briefly, the procedure for installing a foreign command involves the use of a DCL assignment statement to assign the name of the image file to a symbol that is subsequently 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 concerning the procedure for installing a foreign command, refer to the *VAX/VMS DCL Dictionary*.

Example 3–14 shows a program called COMMARG.C, which displays the command-line arguments that were used to invoke it.

Example 3–14: 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:VAXCRTL.OLB RETURN
$ LINK  COMMARG RETURN
```

Sample output from this program 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 that have 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.

3.12 Identifiers

Identifiers can consist of letters, digits, dollar signs ($), and the underscore character (_). You should not create identifiers with a length of more than 31 characters. If you do, the compiler ignores all characters after the thirty-first character.

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.

The dollar sign should be used only within identifiers for VMS global symbols. Identifiers that contain dollar signs may not be portable.

Use 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. For example, the compiler would consider

```
int globalvalue ss$_accvio = 0;

globalvalue SS$_ACCVIO;
```

  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 concerning **globalvalue**, refer to Chapter 7, Storage Classes and Allocation.
- Type all other identifiers and keywords in lowercase.

3.13 Keywords

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 actually 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 3–1 lists the VAX C keywords.

Table 3–1: VAX C Keywords

Keyword	Meaning
Type specifiers:	
int	Integer (On a VAX, 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.
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 global variable.
globalref	Reference to global variable.
globalvalue	Definition or declaration of global value.
readonly	Allocated in read-only program section.
noshare	Assigned NOSHR program section attribute.

Table 3–1 (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 containing loop.
break	Terminates its corresponding **switch** or loop.
if	Executes 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**.
entry	None (reserved for future use).
Operator:	
sizeof	Computes size of 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
```

For more information concerning these identifiers, refer to Chapter 8, Preprocessor Directives.

3.14 Blocks

A block is a compound statement surrounded by braces ({ }). You can use a block wherever 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 3–15 presents nested blocks and the differences in the scope of declared variables.

Example 3–15: 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"    */
❶   int i;
    i = 1;
        .
        .
        .
    if (i == 1)
       {                           /* An inner block           */
❷         float i;
              .
              .
              .
          i = 3e10;
       }
}
```

The following numbers correspond to the numbers in the previous example:

❶ 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**.

❷ 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 concerning data types, refer to Chapter 6, Data Types and Declarations. For more information concerning scope and storage classes, refer to Chapter 7, Storage Classes and Allocation.

3.15 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.

3.16 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 that, combined, offer much of the functionality of LINT.

Feature	Description
/STANDARD=PORTABLE	When you use the CC DIGITAL Command Language command to compile your source code, add this qualifier to CC. The compiler flags any construct 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 3.10 for complete information.
VAXSCA Support	The VAX Source Code Analyzer is a source code cross reference and static analysis tool that you can use with VAX C source code. VAXSCA'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 4

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
- Expressions and blocks as statements
- Conditional statements
- Looping statements
- Interrupting statements

4.1 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.

4.1.1 The null Statement

Null statements are used 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++)
   ;
```

Refer to Section 4.2 and Section 4.4 for more information concerning the statements mentioned here.

4.1.2 The goto Statement

The **goto** statement transfers control unconditionally to a labeled 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 can not exist in storage. Attempts to access such variables can cause a run-time error. For more information concerning the automatic variables, refer to Chapter 7, Storage Classes and Allocation.

4.1.3 The labeled 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 a label is the current function body. Since the label name is independent of the scope rules applied to variables, there can be variables with the same name as the label in the function that contains the label. Labels are used only as the targets of **goto** statements.

4.2 Expressions and Blocks as Statements

The statements in the following subsections are actually expressions or groups of other statements that you can use when the grammar calls for a single statement.

4.2.1 The expression Statement

You can use any valid expression as a statement by terminating it with a semicolon. The following is an example of 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 which can appear in more complex VAX C statements. For more information concerning the valid VAX C expressions, refer to Chapter 5, Expressions and Operators.

4.2.2 The compound Statement

A compound statement in VAX C is sometimes called a block (the compound statement following the parameter declarations in a function definition is called the function body). It allows more than one statement to appear where a single statement is required by the language. The following is an example of a block:

```
{
   int x = 5;

   z = 1;
   if (y < x)
      funct(y, z);
   else
      funct(x, z);
}
```

The block 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 the label of 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 concerning storage classes, refer to Chapter 7, Storage Classes and Allocation.

All function definitions are compound statements.

4.3 Conditional Statements

The statements in the following sections execute only if a tested condition is true.

4.3.1 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 one), 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.

Note that 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 already have an **else** clause.

4.3.2 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 expression, but the result must be type **int**. For more information concerning the data types, refer to Chapter 6, Data Types and Declarations. 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.

At most one statement in the compound statement can have the following label:

```
default :
```

The **case** and **default** labels can occur in any order. When the **switch** statement is executed, the following sequence takes place (note that each case flows into the next unless explicit action is taken, such as a **break** statement):

1. The **switch** expression is evaluated and compared with the constant expressions in the **case** labels.
2. If a **case** label matches the expression's value, the statement or list of statements following that label is 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 4–1.) 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 concerning interrupting statements, refer to Section 4.5.
3. If no **case** label matches the expression's value, 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, the next case encountered is executed.
4. If there is no **case** for the expression's value 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 4–1 uses the **switch** statement to count blanks, tabs, and newlines entered from the terminal.

Example 4–1: Use of 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)
         {
❶           case '\t':  ++number_tabs;
❷                       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);
}
```

The following numbers correspond to the numbers in the previous example:

❶ A series of **case** labels is used to increment the counters.

❷ The **break** statement causes control to go back to the **while** loop every time a counter increments. The program passes control automatically to the **while** loop if none of the counters is incremented.

The program 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
```

by printing the following:

```
Blanks      Tabs  Newlines
     7         2         3
```

If you were to omit the **break** statements, the program would print 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.

4.3.2.1 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 variables are initialized 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);
      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.

4.4 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, this loop is the **for** statement. Some loops evaluate an expression and then execute the body of the loop; in VAX C, this loop is the **while** statement. Some loops execute the loop body and then evaluate the expression, thereby guaranteeing at least one execution of the body; in VAX C, this loop is the **do** statement.

4.4.1 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 particularly 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. The 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 true, 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:

```
expression-1;
while ( expression-2 )
   {
      statement
      expression-3;
   }
```

The VAX C compiler optimizes certain **for** statements for simple loops such as the following:

```
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 always 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 illustrates an infinite loop:

```
for (;;) statement
```

Terminate infinite loops with a **break**, **return**, or **goto** statement.

4.4.2 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 ten, it assigns x to the *x*th element of the array and then increments the variable x. If the expression in parentheses evaluates to false, the loop body never executes.

4.4.3 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.

4.5 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.

4.5.1 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;
```

4.5.2 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;
```

Review the following syntax summary to see the effects of the **continue** statement on the looping statements:

```
goto  label;
```

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( . . . )
 {
   .
   .
   .
  goto label;
   .
   .
   .
  label:
   ;
 }
```

```
   do
{
   .
   .
   .
  goto label;
   .
   .
   .
  label:
   ;
}
while( . . . )
```

```
  for( . . . ; . . . ; . . . )
{
   .
   .
   .
  goto label;
   .
   .
   .
  label:
   ;
}
```

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 which is inside a loop causes continued execution of the enclosing loop after exiting from the body of the **switch** statement.

4.5.3 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 calling 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 concerning the **void** data type and function return values, refer to Chapter 3, Program Structure.

Chapter 5

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. They have no operators and they 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 5.9.1 for more information concerning these rules.

This chapter discusses the following topics:

- lvalues and rvalues
- Primary expressions and operators
- VAX C operators
- Unary expressions and operators
- Binary expressions and operators
- The conditional expression and operator
- Assignment expressions and operators
- The comma expression and operator
- Data type conversions

5.1 lvalues and rvalues

A variable identifier is one of the primary VAX C expressions. (See Section 5.2 for more information concerning primary expressions.) This type of expression yields a single value, the object of the variable. 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. In other words, 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 in the performance of 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:

1. Identifiers of scalar variables, structures, and unions
2. References to scalar array elements
3. References to structure and union members, except for references to fields which are not lvalues
4. References to pointers (also called dereferenced pointers; an asterisk (*) followed by an address-valued expression)
5. Any of the above expressions, enclosed in parentheses

All lvalue expressions represent a single location in a computer's memory. For a pictorial explanation of lvalues and rvalues, refer to Chapter 3, Program Structure.

5.2 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.

5.2.1 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.

5.2.2 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 . . . ".

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

```
result = atof(c);
```

or you can use the identifier **atof** in other contexts without the parentheses, as follows:

```
dispatch(atof);
```

The identifier **atof** converts to the address of that function, and the address is passed to the function dispatch.

5.2.3 Array References ([ ])

Bracket operators ([ ]) are used to refer to elements of arrays. In an array defined as having three dimensions, you 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. For example, consider the statement

```
sample_array[9][4] = 10;
```

This statement assigns a value of ten to the element sample_array[9][4][0]. Consider the following:

```
sample_array = 10;
```

The statement assigns a value of ten 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:

```
funct(array);
```

Bracket operators can also be used to perform general address arithmetic of the form

```
addr[intexp]
```

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.

5.2.4 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 first expression must be an lvalue naming a structure or union. 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 concerning structures and unions, refer to Chapter 6, Data Types and Declarations.

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 first expression must be a pointer to a structure or a 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.

5.3 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 5–1 presents the set of VAX C operators.

Table 5–1: 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	an after increment
++ [postfix]	a++	a before increment
--[prefix]	--a	an after decrement
--[postfix]	a--	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

Table 5–1 (Cont.): VAX C Operators

Operator	Example	Result
=	a = b	a (ith 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)

The 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 conditional operator (a ternary operator), which 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 5.2 for more information).

Table 5–2 presents the precedence by which the compiler evaluates operations. Those 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 5–2: 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

For example, consider the following expression:

```
A*B+C
```

The identifiers A and B are multiplied first because the multiplication operator (*) is of higher precedence than the addition operator (+). The associative rule applies to each row of operators. That is, the expression

```
A/B/C
```

is evaluated as

```
(A/B)/C
```

because the division operator evaluates from left to right.

5.4 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 (!)/
- Increment (++) and decrement (--) variables.
- Find addresses (&) and dereference pointers (*).
- Calculate a one's complement (?~).
- Force the conversion of data from one type to another (the cast operator).
- Calculate the sizes of specific variables or of types (**sizeof**).

5.4.1 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.

The result of the expression

```
! expression
```

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.

5.4.2 Incrementing and Decrementing Variables (++ --)

The object to which the lvalue refers in the expression

```
++lvalue
```

is incremented before its value is used. Upon 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.

The object to which the lvalue refers in the expression

```
lvalue++
```

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.

The objects of the following lvalues point to other variables:

```
--lvalue
lvalue--
```

These pointers decrement not by the integer value 1, but by the length 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 length determined by the addressed object's data type; if declared as an integer, the variable increments or decrements by the value 1.

When using the increment and decrement operators, do not depend upon 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 evaluates first. To avoid ambiguity, increment the variable in a separate statement.

5.4.3 Computing Addresses and Dereferencing Pointers (& *)

Consider the syntax of the following expression:

```
& lvalue
```

The expression results in the lvalue itself. 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 **auto**. The compiler issues no warning 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 use is not recommended for other applications. For more information concerning the manipulation of argument lists, refer to Chapter 3, Program Structure. For more information concerning the VAX Calling Standard, refer to Chapter 10, Mixed-Language Programming.

Since function identifiers and unqualified array identifiers are already lvalues, you cannot apply the address of an 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 address-of token and generates the appropriate error message when the /STANDARD=PORTABLE qualifier is specified.

When an expression evaluates to an address, as in the following,

```
* pointer
```

the address is used to indirectly access the object to which the address refers. Simply, an expression using the indirection operator (*) evaluates to the object pointed to by a pointer or by an address-valued expression.

5.4.4 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.

5.4.5 Forcing Conversions to a Specific Type (Cast Operator)

The cast operator forces the conversion of an operand to a specified scalar data type. 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 identifier type-specifier may also be a structure, union specifier, an **enum** specifier, or a **typedef** tag.

An abstract-declarator in a parameter declaration is a declaration without an identifier or data type keyword:

```
abstract-declarator ::=
        empty
        ( abstract-declarator )
        * abstract-declarator
        abstract-declarator ( )
        abstract-declarator [ constant-expression ]
```

To avoid confusion with the form

```
abstract-declarator( )
```

the abstract-declarator may not be empty as in the following:

```
(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 such operations as

```
(int (*)[]) P1
```

which casts identifier P1 to "pointer to array of **int**." This kind of cast operation in no way changes 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.

5.4.6 Calculating Sizes of Variables and Data Types (sizeof)

Consider the syntax of the following expression:

```
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 declarations of the objects in 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. See Section 5.4.5 for more information concerning the cast operator.

5.5 Binary Expressions and Operators

The binary operators are categorized as follows:

- Additive operators: addition (+) and subtraction (-)
- Multiplication operators: multiplication (*), mod (%), and division (/)
- Equality operators: equality == and inequality (!=)
- Relational operators: less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (> =)
- Bitwise operators: AND (&), OR (|), and XOR (^)
- Logical operators: AND (&&) and OR (||)
- Shift operators: left shift (< <) and right shift (> >)

5.5.1 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 5.9.1.

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 thus obtaining the offset value, and then adds the offset value to the address of the designated element.

A value of any integral type may be subtracted from a pointer or address; in that case, the same conversions apply as for addition.

When you combine 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.

5.5.2 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 is always true:

```
(a/b)*b + a%b = a.
```

5.5.3 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

```
a<b == c<d
```

the result is the value 1, if both relational expressions have the same truth value, and zero if they do not. 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. For example, consider the following:

```
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, true.

5.5.4 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 statement

```
if (a<b<c)...
```

compares the variable c with zero or one (possible results of a <b); it does not mean "if b is between a and c . . . "

5.5.5 Bitwise Operators (& | ^)

These 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 always evaluates all operands. Figure 5–1 illustrates the effects of Boolean algebra when using the bitwise operators.

Figure 5–1: 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	0	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-3071-84

In Boolean algebra, VAX C compares values bit by bit. If using the bitwise AND, and you are comparing a bit value 1 and a bit value 0, the result is 0. When using the bitwise AND, both compared bits must be 1, as shown in the previous figure, 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. Figure 5–1 illustrates the use of all three bitwise operators on two common values.

5.5.6 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 zero (false) or one (true). If the compiler is able to make an evaluation by examining only the left operand, it does not evaluate the right operand. Consider the following expression:

```
E1 && E2
```

The result is one if both its operands are nonzero, or zero if one operand is zero. If expression E1 is zero, E2 is not evaluated. Similarly

```
E1 || E2
```

is one if either operand is nonzero, and zero otherwise. If expression E1 is nonzero, E2 is not evaluated.

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.

5.5.7 Shift Operators (> > < <)

The shift operators (< <) and (> >) take two operands, both of which must be integral. The compiler performs necessary arithmetic conversions on both operands; then, the right operand is converted to **int**, and the type of the result is the type of the left operand. Consider the result of the following:

```
E1 << E2
```

The result is the value of expression E1 shifted to the left by E2 bits. The compiler clears vacated bits. The result of

```
E1 >> E2
```

is the value of expression E1 shifted to the right by E2 bits. The compiler clears vacated bits if E1 is **unsigned**; otherwise, 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 bits.

5.6 Conditional Expression and 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. For example, consider the following:

```
E1  ?  E2  :  E3
```

If expression E1 is nonzero (true), then E2 is evaluated. If E1 is zero (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. Otherwise, if expressions E2 and E3 are address expressions indicating objects of the same type, the result has that type.
3. Otherwise, either 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.

5.7 Assignment Expressions and Operators (= += -= *= /= %= > >= < <= &= ^= |=)

In VAX C, there are 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). An assignment expression such as

```
E1 += E2;
```

is equivalent to

```
E1 = E1 + E2;
```

The expression E1 is evaluated only 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 possibly contain other operators of a lower precedence. For example, the expression

```
a *= b + 1;
```

is the same as

```
a = a * (b + 1);
```

not

```
a = (a * b) + 1;
```

In the simple assignment expression

```
E1 = E2
```

the value of expression E2 replaces the previous object of E1. Another example, the expression

```
a_number[1] += 100;
```

adds 100 to the contents of a_number[1]. The result of the 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 5.9.1.)

The assignment operator (=) can be used 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.

An address may be assigned 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 null pointer is distinguishable (by the equality operators) from a pointer that points to any object.

For the sake of compatibility with other C implementations, VAX C allows certain deviations from the spellings of compound assignment operators shown in Table 5–2. The deviations are as follows:

- When the operators are written in the order shown in Table 5–2, the two characters can be separated by blank spaces. For example,

  ```
  E1 += E2;
  ```

  and

  ```
  E1 + = E2;
  ```

  are identical.
- The operators can also be written with the characters in reverse order, as in the following:

  ```
  E1 =+ E2;
  ```

However, you should avoid the second 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. This placement avoids ambiguities in such cases as

  ```
  E1 =*p;
  ```

  which multiplies the result of expression E1 by the value of p.

- Even with usage that follows the guidelines, it is possible to introduce ambiguities, as in the following:

```
E1 =/*part of a comment...
```

5.8 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

```
R = T = 1,   T += 2,   T -= 1;
```

assigns the value 1 to variable R and the value 2 to variable T.

The type and value of the result of a comma expression are the type and value of the right operand. The operator evaluates 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. For example,

```
f(a, (t=3,t+2), c)
```

calls the function, f, with the arguments a, 5, and c. In addition, variable t is assigned the value 3.

5.9 Data Type Conversions

VAX C performs data type conversions in four situations:

1. When two or more operands of different types appear in an expression (including an assignment).
2. When arguments other than long integers, addresses, or double-precision floating-point numbers are passed to a function.
3. When arguments that do not conform exactly to the parameters declared in a function prototype are passed to a function.
4. When the data type of an operand is deliberately converted by the cast operator. See Section 5.4.5 for more information on the cast operator.

5.9.1 Conversion of 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**), and any of type **float** convert to **double**.
2. Then, if either operand is **double**, the other converts to **double**, and that is the type of the result.
3. Otherwise, if either operand is **unsigned**, the other converts to **unsigned**, and that is the type of the result.
4. Otherwise, both operands must be **int**, and that is the type of the result.

The arithmetic conversions are performed on all arithmetic operands. Note that 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 whenever possible. Whenever 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 = 0xFFFFFF41;
c = i;
```

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 by 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 5.5.1 for more information concerning scaling pointers.

5.9.2 Conversion of 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 5.9.1.) 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 5.4.5 for more information on the cast operator. For more information concerning the manipulation of argument lists, refer to Chapter 3, Program Structure. For more information concerning the VAX Calling Standard, refer to Chapter 10, Mixed-Language Programming.

Chapter 6

Data Types and Declarations

The values of both constants and variables have *data types*. This chapter discusses the following topics in respect to data types:

- Constants
- Variables
- Integers
- Characters
- Pointers
- Floating-point values
- Enumerated types
- Arrays
- Structures and unions
- The **void** keyword
- The **typedef** keyword
- Interpreting variable declarations

6.1 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 concerning symbolic representation of constants, refer to Chapter 8, Preprocessor Directives.) Constants, as do 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.

6.2 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. See Section 6.2.1 for more information concerning data types of variables.

Declarations determine the size of a storage allocation, whereas definitions initiate the allocation of storage. VAX C constants are a good example of the initiation of storage allocation. By design, you can only define and then reference constants; there is no way to declare a constant to claim that the value is defined elsewhere in the program. To define a constant, you give the constant a value. In the case of literal values, you specify the value directly. When using symbolic references, you define an identifier to substitute for a literal value.

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 concerning variable definitions, scope, and storage allocation, refer to Chapter 7, Storage Classes and Allocation.

6.2.1 Classification of Variables

There are two kinds of variables: *scalar* and *aggregate* variables. 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.

6.2.1.1 Data Type Keywords

To declare or define variables, you need to know the VAX C keywords associated with each data type. Table 6–1 lists the VAX C data type keywords according to classification.

Table 6–1: VAX C Data Type Keywords

Scalar Keywords	Aggregate Keywords	Other Type Keywords
int	**struct**	**void**
long	**union**	
unsigned		
short		
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 concerning these type modifiers, refer to Chapter 7, Storage Classes and Allocation.

6.2.1.2 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.

For example, the declaration

```
int  var_number = 10;
```

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 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 was not difficult to interpret, but even experienced VAX C programmers have difficulty interpreting complex variable declarations. See Section 6.12 for more information concerning the interpretation of VAX C variable declarations.

6.3 Integers (int, long, short, char, 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 6–2 specifies the sizes and ranges of integers:

Table 6–2: 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.

6.3.1 Integer Constants

There are three types of integer constants; decimal, hexadecimal and octal. 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, and L 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). On other implementations of the C language, values of the **int** data type require 16 bits of storage. On a VAX, 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. VAX C supports the L suffix only for the sake of program portability.

Integer constants can be specified in decimal, octal, and hexadecimal radixes. An integer constant is assumed to be decimal unless it begins with 0 or 0x; if it begins with zero, 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 could include:

```
133L            /* Long decimal integer          */
0x17A           /* Hexadecimal integer           */
056             /* Octal integer                 */
'a'             /* Decimal 97                    */
'$'             /* Decimal 36                    */
```

Examples of invalid integer constants include:

```
143.                    /* Includes a decimal point          */
3333333333              /* Out of range for int              */
+33333                  /* '+' is an invalid character       */
77af                    /* Hexadecimal constants must be     *
                         *  prefixed with "0x"               */
```

6.3.2 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:

```
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 the 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 four, since you can never be sure if the next byte in the string is a terminating NUL 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 6.8 for more information concerning character-string constants.

6.3.3 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 6–3 presents the escape sequences which specify the nonprinting characters, the apostrophe, and the backslash (\):

Table 6–3: VAX C 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

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 to 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 VT2xx terminal in octal and hexadecimal digits, respectively.

```
'\344'
'\xe4'
```

If the character following the backslash in an escape sequence is illegitimate, the backslash is ignored; that is, the character constant's value is the same as if the backslash were not present.

6.4 Floating-Point Numbers (float, 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.

Table 6–4 specifies the sizes and ranges of real numbers:

Table 6–4: Size and Range of VAX C Floating-Point Numbers

Keyword	Size	Range	Precision
float	32 bits	0.29*10^{−38}$% *to* 1.7 *10^{38}$%	7 decimal digits
double D-Floating	64 bits	0.29*10^{−38}$% *to* 1.7 *10^{38}$%	16 decimal digits
double G-Floating	64 bits	0.56*10^{−308}$% *to* 0.899 *10^{308}$%	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 concerning the compilation command line, refer to Chapter 1, Developing VAX C Programs at DCL Command Level. 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 that will perform type conversions on files, use the VMS Run-Time functions MTHCVT_D_G, MTHCVT_G_D, MTH$CVT_DA_GA, and MTH$CVT_GA_DA if you do not wish to recompile the program. For further information concerning the use of the VMS Run-Time Library, refer to the *VAX/VMS Run-Time Library Routines Reference Manual*

6.4.1 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 E <exponent> , 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
```

6.5 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:

```
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.

Unless a pointer variable is initialized, it is a null pointer. A null pointer is a pointer variable that has been assigned the integer constant 0. However, if you attempt to access data by means of a null pointer. VMS will return 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.

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 address, 4 bytes after px. If px is a pointer to **char**, px + 1 yields the next address, 1 byte after px. The compiler uses the type of the pointed object to scale the arithmetic.

A different result would occur with an expression such as the following:

```
*px + 1
```

This expression evaluates to the value of the object to which px points added to one.

Some contexts may require a pointer of a particular type. This would be necessary, for example, when a function requires that an argument be passed by reference.

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 ampersand (&) operator is used to take the address of an object. For example, consider the following:

```
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.

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. 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 concerning psects, refer to Chapter 7, Storage Classes and Allocation. For more information concerning instances where you would apply the ampersand operator to a constant, refer to Chapter 10, Mixed-Language Programming.

If you do apply the ampersand to **register** variables, the optimizing section of the compiler will prevent any promotion to registers.

If you apply the ampersand to function or array identifiers, VAX C issues a message, since asking for the address of an expression returning an address is redundant.

6.6 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 illustrates 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 zero through four. These enumerators are the constant values of the type shades and can be used wherever constants are valid.

If the tag has already 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 reference 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 one greater than the previous value. For example, consider the following:

```
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. It is not valid to say:

```
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.

Nor is it valid to say:

```
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 concerning the use of these storage class keywords with enumerated types, refer to Chapter 7, Storage Classes and Allocation.

6.7 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:

```
int table_one[10][2];
```

Here, variable table_one is a two-dimensional array containing 20 integers. You can use VAX C operators in forming 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. Omission of 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:

```
int array1[10];
second_function()
{
    .
    .
    .
}
```

For more information concerning external data declarations, refer to Chapter 7, Storage Classes and Allocation.

- If the declaration of the array includes initializers, the size of the array can be omitted. In 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;          /* Incrementor 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.

6.7.1 Initialization of 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:

1. 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.
2. 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 of the elements in the second row which were not specified in the initialization are initialized with zero. For more information concerning the static storage class, refer to Chapter 7, Storage Classes and Allocation.

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.

6.8 Character-String Variables (char *, 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:

```
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";
```

See Section 6.7.1 for more information concerning declaration and initialization of character-string variables.

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 concerning the string copying VAX C RTL functions, refer to the *VAX C Run-Time Library Reference Manual*

6.8.1 Character-String Constants

A character-string constant is a series of characters enclosed in quotation marks (" "). Consider the following:

```
"This is a string constant *** "
```

It has data type of an 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 6.3.2 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.

6.9 Structures and Unions (struct, union)

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 operators that are valid with structures and unions are the simple 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 concerning passing of arguments, especially between programs written in different VMS programming languages, refer to Chapter 10, Mixed-Language Programming.

The difference between structures and unions lies in the way their members are stored:

- The members of a structure all begin at different offsets from the base of the structure. The offset of a particular member corresponds to the order of its declaration; the first member is at offset 0. Each successive nonfield member of a structure begins at the next byte boundary; there is no implicit type alignment. 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.

- In a union, every member begins at offset 0 from the address of the union. The size of the union in memory is the size of 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. You cannot initialize unions. For more information concerning unions, refer to Chapter 3, Program Structure.

6.9.1 Declaring a Structure or Union

Structures and unions are declared with the **struct** or **union** keywords. You can follow the keywords **struct** or **union** 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 structures (or two unions) 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 refer to a structure or union 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 illustrates 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    /* Vacuous declaration refers to this  */
    {...};              /* structure, not to the first one declared. */
}
```

In the 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;
```

6.9.2 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.

For example, 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. In fact, 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 than those in which they were declared, a programming practice not recommended, the compiler assumes that the program is erroneous and 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.

6.9.3 Initialization of Structures

In structure declarations, initializers follow the structure variables, not the members. Separate initializing values with commas; delimit them with braces ({ }). See Section 6.7.1 for more information concerning 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 concerning storage classes, refer to Chapter 7, Storage Classes and Allocation.

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 6–1 shows these initialization rules applied to an array of structures.

Example 6–1: Rules for Initialization of Structures

```
main()
{
   int l, m;
   static struct
      {
         char ch;
         int i;
         float c;
      } ar[2][3] =
❶        {
❷           {
❸              { '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);
         }
}
```

The following numbers correspond to the numbers in the previous example:

❶ You must delimit the initialization of each of the array rows with braces.

❷ You must delimit a structure initialization with braces.

❸ 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       0.000000e+00
[1][1]:          0       0.000000e+00
[1][2]:          0       0.000000e+00
```

6.9.4 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 regular structures or unions except for the following:

- 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.

To illustrate 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 of the intermediate aggregate identifiers, as follows:

```
enclosing_struct.nested_union.nested_struct.d
```

If you attempted to access member d without specifying the intermediate identifiers, then you would be accessing the incorrect offset from the incorrect structure. For instance, if you specified the following:

```
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 illustrates 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, you use the following notation:

```
enclosing_struct.d
```

If you use the following notation:

```
enclosing_struct.nested_union.nested_struct.d
```

unpredictable results occur.

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.

6.9.5 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 declarator (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 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 zero causes the next member (generally another field) to be aligned on a byte boundary.

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:

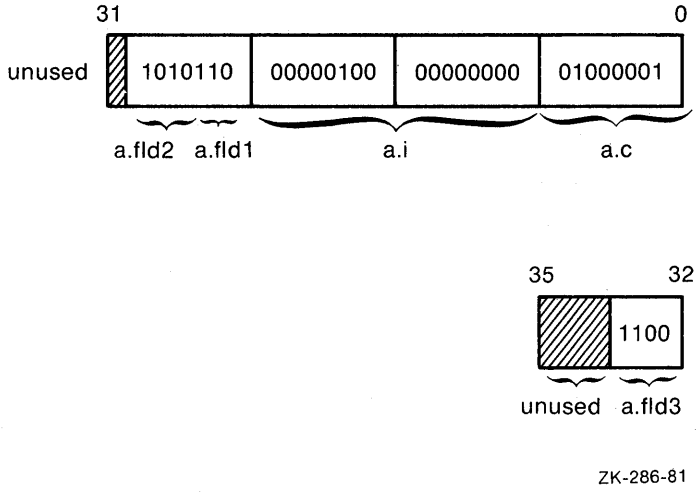
- You cannot declare arrays of fields.
- The address of operator (&) cannot be applied to fields, and consequently there cannot be pointers to fields.

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 6–1 illustrates 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 6–1: Alignment of Structure Members



In Figure 6–1, 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).

6.10 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.

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 concerning the **void** data type and function prototypes, refer to Chapter 3, Program Structure.

6.11 The typedef Keyword

The keyword **typedef** 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 for pointer to character, STRING for 10-element array of characters, and CF for function returning a character. Each of the type definitions can be used in that scope to declare variables, as in:

```
CF      c;              /* "c": Function returning a character  */
STRING  s;              /* "s": 10-character string             */
```

6.12 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 always immediately apparent, even to an experienced C programmer, this section gives guidelines for interpreting and constructing VAX C declarations.

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, such as the following:

```
x  =  *px;
```

In the case of simple declarators, this symmetry makes it fairly 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.

More complicated declarators can be more 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:

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, for 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 fairly 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 scheme[1] for translating operators to English and vice versa:

- "*" == "pointer to"
- "( )" == "function returning"
- "[ ]" == "array of"

For a more interesting example, consider:

```
char  *x()[];
```

The breakdown is:

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 . . . ".

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.

[1] Bruce Anderson, "Type Syntax in the Language C: An Object Lesson in Syntactic Innovation," *SIGPLAN Notices* 15, No. 2 (March 1980).

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:

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.

As a last case, consider:

```
char  (* (*x()) []) ();
```

This means:

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, they should be used in actual declarations for clarity.

Chapter 7

Storage Classes and Allocation

The VAX C language defines a number of storage class keywords which 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 describes the following:

- Scope of an identifier
- Location of storage
- Lifetime of storage allocation
- Internal storage class
- Static storage class
- External storage class
- Global storage class
- Data type modifiers
- Storage class modifiers

7.1 Scope

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. This section explains 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, refer to Chapter 6, Data Types and Declarations.

7.1.1 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 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, refer to Chapter 8, Preprocessor Directives.

The second 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; then, the linker resolves references between these individual modules as well as to external references. For more information concerning compiling and linking, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

7.1.2 Position of the Declaration

In determining 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, whereas 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 6, Data Types and Declarations, before reading the rest of this section.

The location of a declaration establishes the scope of an identifier. If a declaration is located inside of a block (code 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, whereas 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 of the storage class *internal;* 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 storage class external or of the storage class global; 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.

Table 7–1 presents the list of storage classes followed by the storage class specifiers used to establish scope.

Table 7–1: 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**

[1] Functions declared without a storage class specifier are of the external storage class, by default.

For more information concerning the internal storage class, refer to Section 7.3. For more information concerning the static storage class, refer to Section 7.4. For more information concerning the external storage classes, refer to Section 7.5. For more information concerning the global storage classes, refer to Section 7.6.

If you choose, 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 7.7 for more information concerning the data type modifiers. See Section 7.8 for more information concerning the storage class modifiers.

7.1.3 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:

1. Compile the program so that the compiler recognizes all identifiers in the compilation unit, thus avoiding error messages.
2. 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 manual 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 7–2 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 7–2: Scope and the Storage Class Specifiers

	Inside a Function		Outside a Function	
Storage Class	**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

[1] Compilation Unit

Identifier (null) signifies the absence of a storage class specifier from the declaration. The compiler treats a (null) inside of a function or block as an identifier declared with the **auto** keyword; the compiler treats a (null) outside of all functions as an external definition, the identifier being of the external storage class.

In Table 7–2, the notation [**extern**] signifies identifiers of the external storage class. A single definition exists without the use of a storage class specifier; other declarations, that use the **extern** specifier, possibly exist that reference that definition. This notation is used throughout this chapter. See Section 7.5 for more information concerning the external storage class.

7.1.4 Program Example

Determining the scope of **static**, external, and global symbols can be very difficult. In Example 7–1, consider the scope of the following identifiers.

Example 7–1: Scope and Externally Defined Variables

```
Compilation Unit 1
------------------

globaldef int  GLOBAL_1;
          int  EXT_2;
static    int  STAT;

f1()
{
   globaldef int  GLOBAL_2;
      .
      .
      .
}

extern int  EXT_1;

f2()
{
   .
   .
   .
}
```

```
Compilation Unit 2
------------------

int  EXT_1;

f3()
{
   extern int  EXT_2;
      .
      .
      .
}

globalref int  GLOBAL_2;

f4()
{
globalref int  GLOBAL_1;
   .
   .
   .
}

f5()
{
   static int  STAT;
   .
   .
   .
}
```

The following list specifies the variable identifiers in the previous example, and in which functions they can be accessed without compile-time errors:

Identifier	Scope
GLOBAL_1	This variable is defined outside of all functions in compilation unit number 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). In compilation unit number 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.
GLOBAL_2	This variable is defined inside of the function f1. In compilation unit number 1, the scope of GLOBAL_2 only extends from the declaration of GLOBAL_2 to the end of function f1. In compilation unit number 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).
EXT_1	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 number 1, you can access EXT_1 in the function f2 (from the point of the declaration to the end of the compilation unit). In compilation unit number 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).
EXT_2	This variable is defined outside of all functions. In compilation unit number 1, you can access EXT_1 in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit). In compilation unit number 2, the declaration of this variable is located inside of the function f3; you can access EXT_1 from the location of this declaration to the end of function f3.

Identifier	Scope
STAT	There are two variables with the same name but with different permanent storage locations. In essence, these are two different variables. In compilation unit number 1, the variable is defined outside of all functions. You can access STAT, in compilation unit number 1, in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit). In compilation unit number 2, the separate variable is defined inside of the function f5; you can access this variable STAT from this declaration to the end of the function f5.

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 number 2 in the previous example, identifier EXT_2 is made private to function f3 simply by placing the declaration inside of the function body. If you want to keep a variable private to compilation unit number 1, you can 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** assures privacy to the function, since these specifiers cannot be used outside of 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.

7.2 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 which 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 7–3 shows the location and lifetime of a variable when you use each of the storage class keywords:

Table 7–3: 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

When working with some of the storage class keywords, you need to know about the programs sections (or, *psects*) that are created by your data declarations and by VAX C. For detailed information concerning psects and the VAX C storage classes, refer to Chapter 11, VAX C Implementation Notes.

7.3 Internal Storage Class

You can assign the internal storage class to identifiers using the **auto** and **register** storage class specifiers. These specifiers are described in the following sections.

7.3.1 The auto Specifier

You use the **auto** storage class specifier to define a variable whose storage is allocated automatically 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 concerning the **switch** and **goto** statements, refer to Chapter 4, Statements.

Within a function, **auto** is the default storage class. That is, any variable (other than a function name) that is 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 can assign **auto** variables to machine registers, if possible. Otherwise, they are placed on the run-time stack.

Example 7–2 illustrates the reinitialization of two **auto** variables with the same name.

Example 7–2: Reinitialization of 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);
}
```

The following numbers correspond to the numbers in the previous example:

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.

Output from this program 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. The two variables do not conflict, however. While the **for** loop is executing, the variable x declared inside the block supersedes the variable x declared outside the block.

7.3.2 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 of 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 other use of the variable that would make it inappropriate for assignment to a register, the variable will not be flagged. For example, if the compiler detects the application of the address-of operator (&) to a variable which was declared with the **register** specifier, the variable is not placed in a register.

7.4 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 of 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 can not 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. A **static** function is not known to the linker and can be referenced only from within its defining module.

For more information concerning the possible combinations of specifiers and modifiers, and the effects of the storage class modifiers on program section attributes, refer to Chapter 11, VAX C Implementation Notes.

7.5 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 of 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 of 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. Also, if needed, you can 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 always 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. However, this is not good programming practice, and if used, your programs might 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. Therefore, it is good programming practice to express all external variables (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 concerning the possible combinations of specifiers and modifiers, and the effects of the storage class modifiers on program section attributes, refer to Chapter 11, VAX C Implementation Notes.

7.6 Global Storage Class

You can assign the global storage class to identifiers using the **globaldef**, **globalref**, or **globalvalue** storage class specifiers. These specifiers are described in the following sections.

7.6.1 The globaldef and globalref Specifiers

You use the **globaldef** specifier in the definition of a global variable; you use the **globalref** specifier in reference to a global variable defined elsewhere in the program. The specifiers **globaldef** and **globalref** are used in much the same way as with external storage class. Simply, use **globalref** to refer to storage allocated elsewhere by a **globaldef** declaration.

When defining a global symbol using the **globaldef** specifier, you can 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 concerning the possible combinations of specifiers and modifiers, and the effects of the storage class modifiers on program section attributes, refer to Chapter 11, VAX C Implementation Notes.

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 7–3 illustrates the use of global variables.

Example 7–3: Use of 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);
}
```

The following numbers correspond to the numbers in the previous example:

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.

❷ The variable velocity is a variable of storage class **globaldef** and is stored in the psect $DATA.

❸ The variable miles is also a variable of storage class **globaldef** but is stored in the user-specified psect "distance."

❹ When the variable velocity prints after the function fn executes, the value will have changed. Global variables have only one storage location.

❺ 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 this program 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
```

7.6.1.1 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 concerning multi-language programming, refer to Chapter 10, Mixed-Language Programming.

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 differences between the external and global variables. For example:

- The global variables correspond to global symbols declared in assembly language programs whereas external variables correspond with FORTRAN common blocks.
- If you have a limited amount of storage available, you may prefer to use the **globalvalue** specifier (see Section 7.6.2) since it does not occupy storage in your program if expressible in 32 or fewer bits; the external variables always create program sections (psects).
- You can declare a global variable, using **globaldef**, inside of a function or block, and by using a **globalref** specifier, access the identifier from another compilation unit. With external variables, you must always define the variable outside of all functions and blocks, and then access that variable in other compilation units by use of **extern** declarations.
- 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.

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, you should express all global and external variable identifiers in uppercase letters.

Another similarity between the external and global storage classes is that the external variables (by default) and the global variables (optionally) can be placed 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 concerning the possible combinations of specifiers and modifiers, and the effects of the storage class modifiers on program section attributes, refer to Chapter 11, VAX C Implementation Notes.

7.6.2 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 the definition of 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 7–4 illustrates the use of the **globalvalue** storage class specifier.

Example 7–4: Using the globalvalue Specifier

```
/*  This program illustrates references to previously defined  *
 *  globalvalue symbols.                                       */

globalvalue FAILURE = 0, EOF = -1;

main()
{
   char c;
                                  /* Get a char from stdin    */
   while ( (c = getchar())  !=  EOF)
      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    */
   if ( (isalnum(param_c))  !=  FAILURE)
      printf("%c\n", param_c);
   return;
}
```

In the previous program, 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. You should express these symbols in uppercase.

7.6.3 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 **globalvalue**s of the same names as shown in the following example.

In the first compilation unit:

```
globaldef enum  light { dim, medium=3, bright }  light_val;

main()

{
    light_val = dim;
    fnlv();
}
```

In the second compilation unit:

```
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 **globalvalue**s dim, medium, and bright.

The **globalref** declaration in the second compilation unit allows the enumerated constants to be used as **globalvalue**s. That is, the constants can be referenced, but not initialized.

For more information concerning the enumerated type, refer to Chapter 6, Data Types and Declarations.

7.7 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.

7.7.1 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 of 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 will either interpret 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 this next 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 concerning psects and the VAX C storage classes, refer to Chapter 11, VAX C Implementation Notes. 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 and therefore 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 becomes 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 will automatically cause the pointer to become constant. Therefore, DIGITAL recommends that you do not mix constant and nonconstant variables in a **globaldef** declaration that names a psect, as your program may generate unpredictable results.

7.7.2 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 of 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.

7.8 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. An example is as follows:

```
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.

7.8.1 The noshare Modifier

The storage class modifier **noshare** assigns the attribute NOSHR to the program section of the variable. This modifier is used when you wish 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 would 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.

The modifier **noshare** can be used with the storage class specifiers **static**, [**extern**], **globaldef**, and **globaldef**{"name"}. For more information concerning the possible combinations of specifiers and modifiers, and the effects of the storage class modifiers on program section attributes, refer to Chapter 11, VAX C Implementation Notes.

You can use **noshare** alone; when you do this, an external definition of storage class [**extern**] is implied. 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**.

7.8.2 The readonly Modifier

The storage class modifier **readonly**, 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.

Both the the **readonly** and **const** modifiers can be used 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 illustrates 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.

7.8.3 The _align Modifier

The **_align** modifier allows you to align objects of any of the VAX C data types on a specified storage boundary. You 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 either 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 two. The power of two tells VAX C the number of bytes to pad in order to align the data. So, in the previous example, integer 3 specifies an alignment of 2^3 bytes, which is 8 bytes—a quadword of memory.

The following list presents all of the predefined alignment constants, their equivalent power of two, and their equivalent number of bytes:

Constant	Power of Two	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

Chapter 8

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.

Those interested in porting programs to and from other C implementations should take care in choosing which preprocessor directives to use within their programs. See Section 8.3 for more information concerning conditional compilation. For a complete discussion of portability concerns, refer to the *VAX C Run-Time Library Reference Manual.*

The preprocessor directives are introduced by number signs (**#**) that *must* appear in column one of the source listing. This chapter discusses the following preprocessor directives:

- **#define**, **#undef**—Defines token replacements (including preprocessor macro substitutions).
- **#dictionary**—Extracts Common Data Dictionary data definitions and includes them in the source file.
- **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, **#endif**, and the **defined** operator—Control under which conditions segments of code are to be compiled or not.
- **#include**—Includes source text from an external file or library.
- **#line**—Specifies a new line number and file name at the terminal, not in the listing file.

- **#module**—Specifies a module name to the VMS Linker.
- **#pragma**—Performs an implementation-specific task.

This chapter also discusses the following VAX C predefined macros:

- __DATE__ (evaluates to the current date).
- __FILE__ (evaluates to the name of the program source file).
- __LINE__ (evaluates to the line number in the source file).
- __TIME__ (evaluates to the current time).

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 concerning the compilation unit, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

8.1 Token Definitions (#define, #undef)

The **#define** directive specifies a token string that 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 token.

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 8–1 illustrates nested **#define** directives.

Example 8–1: 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;
}
```

When you compile this example program with the command

```
$ CC/LIST/SHOW=INTERMEDIATE EXAMPLE RETURN
```

the following listing results:

```
 1                    /*  Show multiple substitutions and
                         listing format             */
 2
 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;
                 1    writer = james + LAST;
                 2    writer = james + michener;

11       1
12       1            #define LAST joyce
13       1            writer = AUTHOR;
                 1    writer = james + LAST;
                 2    writer = james + joyce;
14       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 is the final text that the compiler processes:

```
writer = james + joyce;
```

The **#define** directive may be continued onto subsequent lines if necessary. You must end each line to be continued with a backslash (\). 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 after the identifier being defined, or anywhere after the left parenthesis in a macro definition.

Comments within the definition line can be continued without the backslash/newline. In the following example, all of the text must appear on the same line unless comments appear in the <white-space> :

```
#<white-space>define<white-space>identifier[(]
```

The optional left parenthesis begins a macro parameter list (see Section 8.1.2), and it *must not* be separated from the identifier.

8.1.1 Constant Identifiers

The first form of the **#define** directive defines a simple substitution, usually of a constant for a frequently used identifier. A common use of the directive is to define the end-of-file (EOF) indicator:

```
#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, as in

```
i=EOF;
```

If the token string –1 is substituted for the identifier EOF, then the contiguous characters (=–) may be mistaken for an operator.

8.1.2 Macro Substitutions

Macros are text substitutions that include a list of parameters. You call macros the same way you call a function. The parameters are replaced by the corresponding arguments and the text is inserted into the program stream. 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 syntax of a macro definition is as follows:

```
#define name([parm1[,parm2,...]]) [token-string]
```

The identifiers' name, parm1, parm2, and so forth are identifiers, and the identifier token-string is arbitrary text.

After the macro definition, all macro references in the source code with the following form

```
name([arg1[,arg2,...]])
```

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.

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 actual 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 concerning the conditional or bitwise operators, refer to Chapter 5, Expressions and Operators.

Preprocessor directives and the macro references have syntax that is independent of the VAX C language. The following list gives the rules for the specification of 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:

```
#ΔdefineΔname(Δparm1Δ,Δparm2Δ)Δ\
Δtoken-stringΔ
```

- 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 the specification of macro references:

- Comments and white space characters (spaces, horizontal and vertical tabs, carriage returns, new lines, and form feeds) may be used freely within a macro reference. In particular, they may appear anywhere that the delta symbol appears in the following example:

```
nameΔ(Δarg1Δ,Δarg2Δ)
```

- 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, although 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, a parameter should be delimited 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, you should not pass the following argument to the **_toupper** macro:

```
_toupper(p++)
```

When the argument p++ is substituted in the macro definition, the resultant line of code within the program stream is the following:

```
((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 a good programming practice. Even if you are fully aware of possible side effects, the token strings within macro definitions are easily changed, thereby changing the side effects without warning.

8.1.3 Listing of Substituted Lines

The /SHOW command line qualifier has two optional values that enable the listing of all lines that have been modified by macro substitutions. The values are EXPANSION and INTERMEDIATE.

With the qualifiers

```
/LIST/SHOW=EXPANSION
```

the listing produced by the compiler 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.

With the qualifiers

```
/LIST/SHOW=INTERMEDIATE
```

the compiler lists all intermediate substitutions with one substitution per line.

Without one of these two qualifiers, the compiler only lists the original form of a line.

Example 8–1 demonstrates the effect of the /SHOW=INTERMEDIATE qualifier. For more information concerning the format of VAX C compiler listings, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

8.1.4 Canceling Definitions (#undef)

The following directive

```
#undef identifier
```

cancels a previous definition of the identifier by **#define**.

8.2 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 salesmen might have subdirectories for records such as salary and commission history or personnel history.

The advantages of using the CDD include the following:

- 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 concerning the CDD, refer to the *VAX Common Data Dictionary Reference Manual*, the *VAX Common Data Dictionary Utilities Manual*, and the *VAX Common Data Dictionary Language Reference Manual*.

8.2.1 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

```
#dictionary cdd_path
```

The *cdd_path* is a character string that gives the pathname of a CDD record or a macro which expands to the pathname of the record. For example:

```
#dictionary "CDD$TOP.personnel.service.salary_record"
```

This pathname describes all subdirectories leading to the salary_record data definition, beginning with the root directory (CDD$TOP).

The logical name CDD$DEFAULT can be used to define a default pathname for a dictionary directory. This logical name can specify part of the pathname for the dictionary object. For example, you can define CDD$DEFAULT as follows:

```
$ DEFINE CDD$DEFAULT CDD$TOP.PERSONNEL
```

When this definition is in effect, the **#dictionary** directive can contain

```
#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, a definition of a structure containing someone's first and last name could be written 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 were found in a source file named CNAME.DDL, it could be included in the CDD subdirectory named "doc" by issuing the following command:

```
$ CDDL cname
```

After this command has been executed, 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"
```

These lines of VAX C code result 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, with *cname* used 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.

8.2.2 Support for CDD Data Types

The CDD supports all VMS data types. VAX C can translate all of 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 which 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. When 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.

Table 8–1 summarizes the mapping between CDD data types and VAX C data types.

Table 8–1: 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
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** [*n*]
Varying text[3]	Unsupported
Numeric string:	
unsigned	Unsupported
Left separate	Unsupported
Left overpunch	Unsupported
Right separate	Unsupported
Right overpunch	Unsupported
Zoned sign	Unsupported

[1] A 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**.

[3] For these data types, the length of the structure is two bytes longer than the string to allow for the length field.

Table 8–1 (Cont.): Mapping Between CDD and VAX C Data Types

CDD Data Type	C Data Type
Packed decimal string	Unsupported
Bit	Bit field[2]
Bit unaligned	Bit field[2]
Date and time	Unsupported
Date	Unsupported
Virtual field	Ignored
Varying string[3]	Unsupported

[2] A message is issued if the bit string length is greater than 32.

[3] For these data types, the length of the structure is two bytes longer than the string to allow for the length field.

Unsupported data types are mapped into VAX C as structures of character arrays of the appropriate size. The declaration of these data types follows the 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.

8.3 Conditional Compilation (#if, #ifdef, #ifndef, #else, #elif, #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 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, therefore, 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 issues a diagnostic message and treats the identifier as though it were the constant zero.

The **#ifdef** directive has the form:

```
#ifdef identifier
```

This directive checks whether the identifier was previously defined by a **#define** directive.

The **#ifndef** directive has the 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 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 form:

```
#elif  constant-expression
```

The **#elif** line 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 form:

```
#endif
```

This directive ends the scope of the directives.

8.3.1 The defined Operator

If you need to check to see if many tokens are defined, you may wish to use the special **defined** operator in a single use of the **#if** line. In this way, you can check for token definitions in one concise line without having to use many **#ifdef** or **#ifndef** directives.

For example, if you want to check the following tokens:

```
#ifdef  token1
printf( "Oh, Mary!\n" )
#endif

#ifndef  token2
printf( "Oh, Mary!\n" )
#endif

#ifdef  token3
printf( "Oh, Mary!\n" )
#endif
```

you can use the **defined** operator in a single use of the **#if** preprocessor directive, as follows.

```
#if  defined (token1)  ||!defined (token2) ||defined (token3)
printf( "Oh, Mary!\n" )
#endif
```

You can use **defined** as you would any other operator. However, you can only use **defined** in the evaluated expression of an **#if** or **#elif** preprocessor directive.

8.4 File Inclusion (#include)

The **#include** directive inserts external text into the token stream delivered to the compiler. Often, global definitions for use with VAX C Run-Time Library (RTL) 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!/DBA0/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, refer to the *VAX C Run-Time Library Reference Manual.*

8.4.1 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; and if it is, 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, an error occurs. Valid DEC/Shell file specifications are a subset of valid UNIX file specifications. For more information concerning the valid DEC/Shell file specifications, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

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 (< >). When the compiler encounters this form of file inclusion, it translates the user-defined logical name, VAXC$INCLUDE, which you can define to be a valid directory specification or a search list of valid directory specifications; if VAXC$INCLUDE is defined, the compiler searches the directory or directories for the specified file. Before each execution 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$INCLUDE translates to a directory or a search list of directories, and the compiler cannot locate the specified file, the compiler generates an error message. If VAXC$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 concerning search lists, refer to the DCL command DEFINE in the *VAX/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 DEC/Shell path name

```
/sys/file.h
```

and then translates the DEC/Shell path name to the VMS file specification

```
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.

8.4.2 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 first searches the directory containing the compiled source file for the included file, *not* the current RMS default directory. For example, given the current directory, DBA0:[CURRENT], and given the following CC command line

```
$ CC  DBA0:[OTHERDIR]EXAMPLE.C RETURN
```

the compiler searches DBA0:[OTHERDIR] for any included files delimited by quotation marks, even though the current RMS default is the directory, DBA0:[CURRENT].

If the compiler 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 execution of your program, you have the flexibility of redefining C$INCLUDE to be any valid directory or list of directories you choose.

As with the VAXC$INCLUDE, do not define C$INCLUDE to be a rooted directory or subdirectory. You should 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 concerning search lists, refer to the DCL command DEFINE in the *VAX/VMS DCL Dictionary*.

CAUTION

If you include a file from SYS$LIBRARY by using the angle brackets, and if the included file contains a second **#include** line that delimits the file specification with quotation marks, the compiler will search in the directory containing the source file for the specified file, not in SYS$LIBRARY. When nesting

#include directives as described previously, the file specification in quotation marks *must* contain complete device and directory information.

8.4.3 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, because modules within a text library are indexed and thus easier to manipulate than files in a directory. 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.
- 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 CAUTION), 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 in the CC command, or if the specified module cannot be found in any of the specified libraries, the following actions are taken:
  - If the user has 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.

8.4.4 Token Substitution in #include Directives

VAX C allows token substitution within the **#include** preprocessor directive.

For instance, if you wanted to include a file name, you could use the following two directives.

```
#define  token1  "file.ext"

#include token1
```

If you use defined tokens in **#include** directives, the tokens 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
```

8.5 Specification of Line Numbers (#line, #)

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 also specifies a new file specification for the program source file. The **#line** directive will not change the line numbers in your compilation listing, only the line numbers given in messages (for example, error 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:

```
#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.

8.6 Specification of Module Name and Identification (#module)

When you compile source files to create an object file, the compiler assigns to that file the first filename of those specified in the compilation unit. The compiler adds the .OBJ file extension to the object file. Internally, VMS (the debugger and the librarian as well) recognizes the object module by the filename; the compiler also gives the module a V1.0 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 or a character-string constant with no more than 31 characters.

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, as well as in a **#dictionary** 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.

8.7 Implementation-Specific Preprocessor Directive (#pragma)

The **#pragma** directive performs tasks as designated by the particular implementation of the C language. VAX C recognizes the following specification for enabling and disabling the padding of structures and alignment of members:

```
#pragma [no]member_alignment
```

8.8 VAX C Predefined Tokens

The following sections describe the VAX C predefined tokens and macros for use in your programs.

8.8.1 Predefined Tokens

The VAX C compiler defines the following preprocessor substitutions; 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.

```
#define CC$gfloat  <value>
```

You can use this definition if you compile your VAX C program using the /G_FLOAT qualifier. If you specify the /G_FLOAT qualifier, <value> is defined as 1.

```
#define CC$gfloat    <value>
```

You can use this definition if you do *not* compile your VAX C program using the /G_FLOAT qualifier. When you do not specify the /G_FLOAT qualifier, <value> is defined as zero.

The symbols may be used by the VAX C programmer to conditionally compile VAX C programs used on more than one operating system to take advantage of system-specific features. See Section 8.3 for more information concerning the use of the preprocessor conditional compilation directives.

Consider the following example:

```
#if       VAXC
#include  rms           /* Include RMS definitions  */
#endif
```

Similarly, you can use the CC$gfloat preprocessor substitution so that you can assign values to variables of type **double** without causing an error. The VAX C compiler will only substitute 1 for CC$gfloat if you compiled the module using the /G_FLOAT qualifier. Therefore, you can conditionally assign values to variables 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 will determine whether or not to substitute the value 1 for every occurrence of these identifiers in a program; these identifiers are reserved by DIGITAL. However, the effect of these definitions may be removed by explicitly undefining the conflicting name. See Section 8.1.4 for more information concerning undefining. For more information concerning the G_floating representation of the **double** data type, refer to Chapter 6, Data Types and Declarations.

8.8.2 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
```

Note that the first d is a space if dd is less than 10.

The following is an example of the __DATE__ macro:

```
printf("%s",__DATE__);
```

8.8.3 The __TIME__ Macro

The __TIME__ macro evaluates to a string specifying the time at which 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__);
```

8.8.4 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__);
```

8.8.5 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__);
```

Using VAX C Features on VMS

Chapter 9

Using VAX Record Management Services (RMS)

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 the use of the VAX C Run-Time Library functions. In any case, the system that ultimately accesses the files in VMS is VAX Record Management Services (RMS).

This chapter introduces the following RMS topics to the VAX C programmer:

- Sequential files
- Relative files
- Indexed files
- Record access modes
- RMS record formats
- RMS functions
- VAX C and RMS
- RMS example program

The file-handling capabilities of VAX C fall into two distinct categories:

1. The VAX C Run-Time Library functions which, with little or no modification, are portable to other C implementations.

2. The RMS functions, which are not portable to other C implementations but which 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 concerning RMS, refer to the following manuals in the VMS document set:

- *Guide to VAX/VMS File Applications.* This manual contains a general description of the record management services of the VMS operating system, and the file creation and run-time options available.
- *VAX Record Management Services Reference 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.

9.1 RMS File Organization

VAX RMS supports three kinds of file organization:

- Sequential
- Relative
- Indexed

The organization of a file determines the way the 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, refer to the *VAX/VMS Utility Routines Reference Manual.*

9.1.1 Sequential File Organization

Sequential files have consecutive records. There are no empty records separating records that contain data. This organization limits the operations on the file to:

- 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.

9.1.2 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. Consequently, 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.

9.1.3 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 may 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 makes possible 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 already 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.

9.2 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 read sequentially all the indexed records in ascending order of their key values; this method is sometimes called the indexed-sequential access method, (ISAM).

9.3 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.

9.4 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 9–1 lists some of the commonly used functions.

Table 9–1: 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$close	Terminates file processing and closes the file.
	sys$erase	Deletes a file.
Record Processing	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$delete	Deletes a record from a file.
	sys$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. For example, because sys$create is a file-processing function, its syntax would be as follows:

```
int     sys$create(fab)
struct  FAB      *fab;
```

Note that 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, refer to the *VAX Record Management Services Reference Manual.*

Finally, all of the RMS functions return an integer status value. The format of RMS status values follows the standard format described in Chapter 10, Mixed-Language Programming. Since they return a 32-bit integer, you do not need to declare the RMS return status before you use it.

9.5 Writing VAX C Programs Using RMS

VAX C supplies a number of **#include** modules that describe the RMS data structures and status codes. These modules are listed in Table 9–2.

Table 9–2: 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 of the above modules.

Most VAX C programmers simply 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.

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, consult the *VAX Record Management Services Reference Manual.*

The prototype data structures, and the structures which they initialize, are listed in Table 9–3.

Table 9–3: 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$rms_xabrdt	XABRDT	Revision date and time extended attribute block
cc$rms_xabsum	XABSUM	Summary extended attribute block

You need not declare these structures before referencing them; 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
```

where 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 *VAX Record Management Services Reference Manual* for a description of the members in each structure.

9.5.1 Initializing File Access Blocks

The file access block defines the attributes of the file. To initialize a file access block, you 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 allocate RMS structures          *
 *  dynamically.                                               */

#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. For example, the statement

```
fblock.fab$l_xab = &primary_key;
```

assigns the address of the extended attribute block named primary_key to the fab$l_xab member of the file access block named fblock.

9.5.2 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;
        .
        .
        .
}
```

9.5.3 Initializing Extended Attribute Blocks

There is only one extended attribute block structure (XAB), but you can initialize it seven ways. 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. Consider the following example:

```
/*  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;
      .
      .
      .
}
```

9.5.4 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 as you would initialize the other RMS data structures; as in 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;
      .
      .
      .
}
```

9.6 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 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 to be 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, you would go through the following steps:

1. Create a source file. The name of the source file in this example is RMSEXP.C. For more information concerning the creation of source files, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

2. Compile the source file with the command

```
$ CC  RMSEXP RETURN
```

   For more information concerning the compiling process, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

3. Link the program with the command

```
$ LINK  RMSEXP, SYS$LIBRARY:VAXCRTL/LIB RETURN
```

   For more information concerning the linking process, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

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 concerning foreign commands, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

5. Run the program, using the 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 9–1 shows the external data declarations and definitions.

Example 9–1: External Data Declarations and Definitions

```
   /* This segment of RMSEXP.C contains external data        *
    * definitions                                            */

❶  #include rms
   #include stdio
   #include ssdef

❷  #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)

❸  struct  FAB fab;
   struct  RAB rab;
   struct  XABKEY primary_key,alternate_key;

❹  struct
      {
         char     ssn[SIZE_SSN], last_name[SIZE_LNAME];
         char     first_name[SIZE_FNAME],
                  comments[SIZE_COMMENTS];
      } record;

❺  char  response[BUFSIZ],*filename;

❻  int  rms_status;
```

The following numbers correspond to the numbers in the previous example:

❶ 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.

❷ 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.

❸ Static storage for the RMS data structures is declared. The file access block, record access block, and extended attribute blocks are defined by the *rms* module. One extended attribute block is defined for the primary key and one is defined for the alternate key.

❹ The records in the file are defined using a structure with four fields of character arrays.

❺ The BUFSIZ constant is used to define the size of the array that will be used to buffer input from the terminal. The filename variable is defined as a pointer to **char**.

❻ 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 9–2, controls the general flow of the program.

Example 9–2: Main Program Section

```
   /* This segment of RMSEXP.C contains the main function    *
    * and controls the flow of the program.                  */

❶  main(argc,argv)
   int   argc;
   char  **argv;
   {
❷     if (argc < 1 || argc > 2)
         printf("RMSEXP - incorrect number of arguments");
      else
         {

            printf("RMSEXP - Personnel Database \
   Manipulation Example\n");

❸           filename = (argc == 2 ? *++argv : "personnel.dat");
❹           initialize(filename);
❺           open_file();

            for(;;)
               {
❻                 printf("\nEnter option (A,D,P,T,U) or \
   ? for help :");

                  gets(response);
                  if (feof(stdin))
                     break;
                  printf("\n\n");
```

(Continued on next page)

Example 9–2 (Cont.): Main Program Section

```
❼               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");

                      case '?': case '\0':
                                           type_options();

                   }
             }

❽         rms_status = sys$close(&fab);

❾         if (rms_status != RMS$_NORMAL)
                      error_exit("$CLOSE");
       }
  }
```

The following numbers correspond to the numbers in the previous example:

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 (filename).

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 9–3 shows the function that initializes the RMS data structures. Refer to the RMS documentation for more information about the file access block, record access block, and extended attribute block structure members.

Example 9–3: Function Initializing RMS Data Structures

```
/* This segment of RMSEXP.C contains the function that    *
 * initializes the RMS data structures.                   */

initialize(fn)  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       ";
```

(Continued on next page)

Example 9–3 (Cont.): Function Initializing RMS Data Structures

```
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 \
                ";
}
```

The following numbers correspond to the numbers in the previous example:

❶ The prototype cc$rms_fab initializes the file access block with default values. Some members have no default values; they must always be initialized. Such members include the file-name string address and size. Other members can be initialized to override the default values.

❷ This statement initializes the file-processing options member with the create-if option. A file will be created if one does not exist.

❸ This statement initializes the record attributes member with the carriage-return control attribute. Records will be terminated with a carriage return/line feed when they are printed on the printer or displayed at the terminal.

❹ 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.

❺ The prototype cc$rms_xabkey initializes an extended attribute block for one key of an indexed file.

❻ The position of the key is specified by subtracting the offset of the member from the base of the structure.

❼ A separate extended attribute block is initialized for the alternate key.

❽ 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).

❾ The key-name member is padded with blanks because it is a fixed-length 32-character field.

Example 9–4 shows the internal functions for the program.

Example 9–4: Internal Functions

```
/* This segment of RMSEXP.C contains the functions that   *
 * control the data manipulation of the program.          */

open_file()
{
❶   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");

❷   rms_status =  sys$connect(&rab);
    if (rms_status != RMS$_NORMAL)
       error_exit("$CONNECT");
}

❸ 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");
```

(Continued on next page)

Example 9–4 (Cont.): Internal Functions

```
    printf("U     Update employee specified by SSN.\n\n");
    printf("?     Type this text.\n");
    printf("^Z    Exit this program.\n\n");
}

❹ 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.   */

❺ error_exit(operations)
char *operation;
{
    printf("RMSEXP - file %s failed (%s)\n",
            operation, filename);
    exit(rms_status);
}
```

The following numbers correspond to the numbers in the previous example:

❶ 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.

❷ The RMS **sys$connect** function associates the record access block with the file access block.

❸ 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.

❹ For each field in the record, the pad_record function fills the remaining bytes in the field with blanks.

❺ 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 9–5 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 9–5: 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);
```

(Continued on next page)

Example 9–5 (Cont.): Utility Function: Adding Records

```
❷    pad_record();

❸    rab.rab$b_rac =  RAB$C_KEY;
     rab.rab$l_rbf =  &record;
     rab.rab$w_rsz =  RECORD_SIZE;

❹    rms_status =  sys$put(&rab);

❺    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");
}
```

The following numbers correspond to the numbers in the previous example:

❶ 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.

❷ When all fields have been entered, the pad_record function pads each field with blanks.

❸ Three members in the record access block are initialized prior to 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.

❹ The RMS sys$put function writes the record to the file.

❺ 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.

Example 9–6 shows the function that deletes records. This function is called when 'd' or 'D' is entered in response to the menu.

Example 9–6: Utility Function: Deleting Records

```
/* This segment of RMSEXP.C contains the function that    *
 * deletes a record from the file.                        */

delete_employee()
{
   int i;
❶  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");
         }
}
```

The following numbers correspond to the numbers in the previous example:

❶ A **do** loop prompts the user 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.

❹ The RMS **sys$find** function locates the record specified by the social security number entered from the terminal.

❺ 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.

❻ The RMS **sys$delete** function deletes the record. The return status is checked only for success.

The type_employees function in Example 9–7 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 9–7: Utility Function: Typing the File

```
/* This segment of RMSEXP.C contains the function that    *
 * displays a single record at the terminal.              */

type_employees()
{
❶   int  number_employees;

❷   rab.rab$b_krf =  1;

❸   rms_status =  sys$rewind(&rab);
    if (rms_status != RMS$_NORMAL)
       error_exit("$REWIND");

❹   printf("\n\nEmployees (Sorted by Last Name)\n\n");
    printf("Last Name      First Name          SSN      \
          Comments\n");

    printf("---------      ----------          ---------\
          --------\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;

          printf("%.*s%.*s%.*s%.*s\n",
                  SIZE_LNAME, record.last_name,
                  SIZE_FNAME, record.first_name,
                  SIZE_SSN, record.ssn,
                  SIZE_COMMENTS, record.comments);
       }

❼   if (number_employees)
       printf("\nTotal number of employees = %d.\n",
                 number_employees);
    else
       printf("[Data file is empty.]\n");
}
```

The following numbers correspond to the numbers in the previous example:

❶ A running total of the number of records in the file is kept in the number_employees variable.

❷ The key of reference is changed to the alternate key so that the employees are displayed in alphabetical order by last name.

❸ 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.

❹ A heading is displayed.

❺ Sequential record access is specified, and the location and size of the record is given.

❻ 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.

❼ 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 9–8 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 9–8: 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;

         fprintf(fp, "%.*s%.*s%.*s%.*s",
                 SIZE_LNAME,record.last_name,
                 SIZE_FNAME,record.first_name,
                 SIZE_SSN,record.ssn,
                 SIZE_COMMENTS,record.comments);
      }
```

(Continued on next page)

Example 9–8 (Cont.): Utility Function: Printing the File

```
❼    if (number_employees)
        fprintf(fp, "Total number of employees = %d.\n",
                number_employees);
     else
        fprintf(fp,"[Data file is empty.]\n");

❽    fclose(fp);
        printf("[Listing file\"personnel.lis\"spooled to \
SYS$PRINT.]\n");
}
```

The following numbers correspond to the numbers in the previous example:

❶ 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**, thus associating the file with the file pointer, fp.

❷ The key of reference for the indexed file is the primary key.

❸ The **sys$rewind** function positions the file at the first record. The return status is checked for success.

❹ A heading is written to the sequential file using the standard I/O function **fprintf**.

❺ 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.

❻ 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.

❼ The number-employees counter is checked. If it is zero, a message is printed indicating that the file is empty. If it is not zero, the total is printed at the bottom of the listing.

❽ 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 9–9 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 9–9: Utility Function: Updating the File

```
/* This segment of RMSEXP.C contains the function that     *
 * updates the file.                                       */

update_employee()
{
   int i;
❶  do
      {
         printf("(UPDATE) 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$C_KEY;
   rab.rab$l_ubf =  &record;
   rab.rab$w_usz =  RECORD_SIZE;

❹  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");
```

(Continued on next page)

Example 9–9 (Cont.): Utility Function: Updating the File

```
❺       else
           {
              printf("Enter the new data or <RET> to leave \
  data unmodified.\n\n");

              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);

❻             pad_record();

❼             rms_status =  sys$update(&rab);
              if (rms_status != RMS$_NORMAL)
                 error_exit("$UPDATE");

              printf("[Record has been successfully \
  updated.]\n");
           }
  }
```

The following numbers correspond to the numbers in the previous example:

❶ A **do** loop prompts for the social security number and places the response in the response buffer.

❷ The response is padded with blanks so that it will correspond to the field in the file.

❸ 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.

❹ 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.

❺ For each field (except the social security number, which cannot be changed), the program displays the current value for that field. If the user presses the RETURN key, the record is placed in the record structure unchanged. If the user makes a change to the record, the new information is placed in the record structure.

❻ The fields in the record are padded with blanks.

❼ 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.

Chapter 10

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 or prewritten system routines from VAX C. You can call any one of the following routine types from VAX C:

- Routines written in other VAX languages
- VMS Run-Time Library routines
- VMS system services

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, whereas 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 Run-Time Library routines and system services. System routines, which are discussed later in this chapter, are documented in detail in the *VAX/VMS Run-Time Library Routines Reference Manual* and the *VAX/VMS System Services Reference Manual.*

10.1 The VAX Procedure Calling and Condition Handling Standard

The VAX Procedure Calling and Condition Handling Standard describes the concepts used by all VAX languages for invoking routines and passing data between them. The following attributes are specified by the VAX Procedure Calling and Condition Handling Standard:

- Register usage
- Stack usage
- Function value return
- 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 VAX/VMS System Routines*.

10.1.1 Register and Stack Usage

The VAX Procedure Calling and Condition Handling Standard defines several registers and their uses, as listed in Table 10–1.

Table 10–1: 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 value return 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 LIFO (last-in/first-out) 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:

- 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.

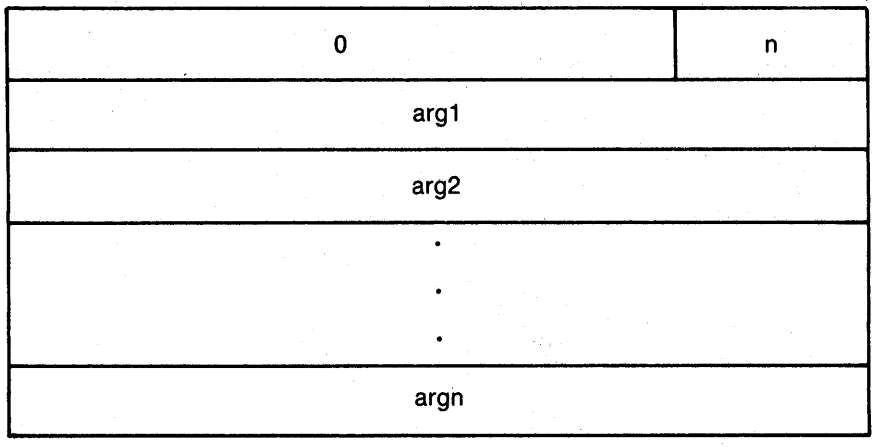
10.1.2 Return of the Function Value

A function is a routine that returns a single value to the calling routine. The *function value* represents the return value that is assigned to the function's identifier during execution. 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.

10.1.3 The Argument List

The VAX Procedure Calling and Condition Handling Standard also defines a data structure called the argument list. You use an argument list to pass information to a routine and receive results. An *argument list* is a collection of longwords in memory that represents a routine parameter list and possibly includes a function value. Figure 10–1 shows the structure of a typical argument list.

Figure 10–1: Structure of a VAX Argument List

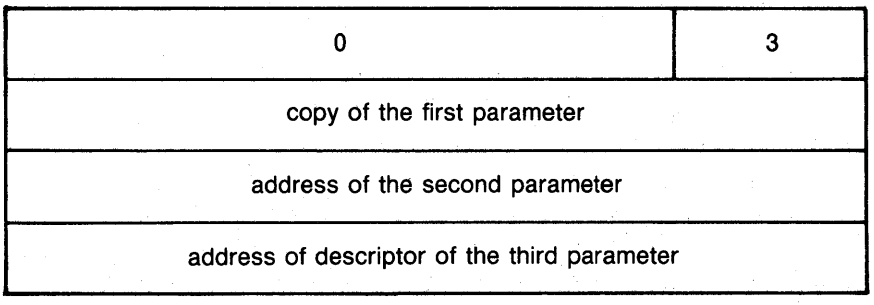


The first longword must be present; this longword stores the number of arguments (the argument count: *n*) as an unsigned integer value in the low byte of the longword. The remaining 24 bits of the first longword are reserved for use by DIGITAL and should be zero. The longwords labeled *arg1* through *argn* are the actual parameters, which can be any of the following:

- 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 10–2 shows this argument list.

Figure 10–2: Example of a VAX Argument List



For additional information on the VAX Procedure Calling and Condition Handling Standard, see the *Introduction to VAX/VMS System Routines.*

10.2 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 that data can be passed in an argument list. When you code a reference to a non-VAX C procedure, you must know how each argument should be passed and write the function reference accordingly.

The three argument-passing mechanisms are described in the following list:

- 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. The VAX C ampersand operator (&) is used to pass the address of an argument.

- 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 (&).

NOTE

In the C programming language environment, you can take the address of an argument and then 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, refer to *VAX/VMS Run-Time Library Routines Reference Manual.*

The following sections outline each of these parameter-passing mechanisms in more detail.

10.2.1 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 at the programmer's 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 or function name in an argument list always results in passing the address of the array or function.

In the special case of argument lists, VAX C allows the ampersand operator to be used on constants as well. However, 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 following status values, expressed here as global symbols:

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 10–1 shows a call to the SYS$READEF system service from a VAX C program.

Example 10–1: 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      */
```

(Continued on next page)

Example 10–1 (Cont.): Passing Arguments by Reference

```
    return_status =  SYS$READEF(event, &cluster_status);
                                        /*  Check for successful    *
                                         *   call                   */
    if (return_status != SS$WASCLR && return_status != SS$WASSSET)
        {
            /* Handle the error here.                               */
                .
                .
                .
        }
    else
        {
            /*  Check bits of interest in CLUSTER_STATUS here.      */
                .
                .
                .
        }
}
```

10.2.2 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, you perform the following steps:

1. Write a structure declaration that models the required descriptor. This involves including the text library module *descrip* 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.

In default cases, VAX C never passes arguments by descriptor. For example, when structure or union names are written 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.

There are several classes of descriptor. Each class requires that certain bits be set in the first longword of the descriptor. For more information concerning the descriptors and their formats, refer to the *Introduction to VAX/VMS System Routines*. In accordance with the information in the handbook, descriptors can be modeled 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                 */
      char  *dsc$a_pointer           /*  Has address of first  *
                                      *   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. The valid class codes are as follows:

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	—
10	DSC$K_CLASS_NCA	—
11	DSC$K_CLASS_VS	—
12	DSC$K_CLASS_VSA	—
13	DSC$K_CLASS_UBS	—
14	DSC$K_CLASS_UBA	—

Class Code	Symbolic Name	Descriptor Class
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 class codes listed in the following table are supported by VAX C; all others are not directly supported by the language. Refer to the *Introduction to VAX/VMS System Routines* manual for a complete list of atomic class codes. The atomic class codes supported by VAX C are as follows:

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 the normal VAX C programming practices. You must define a structure and assign the data's address to the pointer member. You must also assign appropriate values to the members dscw_length, dscb_dtype, and dsc$b_class. For the specific requirements of each descriptor class, refer to the *Introduction to VAX/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 10–2 shows a call to this system service from a VAX C program.

In the previous example, 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 **#include** text library module *descrip*. 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. The $DESCRIPTOR macro may be used 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 10–2: 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$SETPRN             */
                                    /*  Name the descriptor     */
   struct  dsc$descriptor_s  name_desc;

   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);
      .
      .
      .
}
```

10.2.3 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. 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 with identical cases. The documentation uses uppercase as a convention for referring to system service calls to highlight them in the text and examples. Consider the following example:

```
/*  Declare the function as a function returning type INT      */

int  SYS$SETEF();
```

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.

In the declaration of external functions, the VAX C syntax does not indicate the number or types of the arguments, nor does VAX C compare the types of arguments with the types that the system service requires. It is your responsibility to ensure that the argument list of an external function reference contains valid arguments.

In the *VAX/VMS System Manager's Reference Manual*, 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 following symbolic constants.

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 VMS. 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:

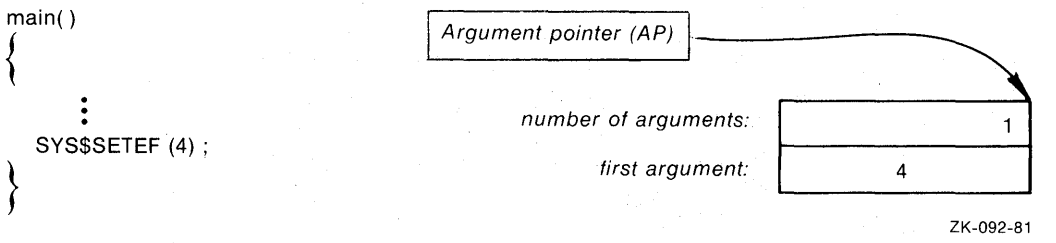
```
enum  cluster0  {completion, breakdown, beginning}  event;
```

Once the flag numbers have been defined, the SYS$SETEF service can be called with the following code:

```
    .
    .
    .
int  status;
event = completion;
    .
    .
    .
status = SYS$SETEF(event);          /* Set event flag          */
    .
    .
    .
```

Figure 10–3 shows an argument being passed by immediate value; in this case, the event flag number passed to SYS$SETEF.

Figure 10–3: Passing Arguments by Immediate Value



10.2.4 Passing Floating-Point Arguments by Immediate Value

Since argument lists consist of longwords, the calling standard dictates that immediate-value arguments be expressible 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). 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 that avoids the promotion of **float** arguments. One such method is to use a structure, as shown in Example 10–3.

Example 10–3: Passing Floating-Point Arguments by Immediate Value

```
/*  This program shows how to pass a floating-point value,      *
 *  in a structure, to avoid the promotion of 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();

main()
{
                                  /*  To hold result of         *
                                   *   OTS$POWRJ                */
   float  result;
   int    power;                  /*  Power argument            */

                                  /*  Structure used to pass    *
                                   *   floating-point var       *
                                   *   by value                 */
   struct { float f; } base;

   base.f =  3.145;               /*  Assign constant to base */
   power  =  2;
   result =  OTS$POWRJ(base, power);

   printf("Result= %f\n", result);
}
```

By default, structures, like everything else, are passed by immediate value. Thus, in Example 10–3, the argument is not interpreted as a **float** and is not promoted to **double**.

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.

The great majority of run-time functions that operate on floating-point values take their arguments by reference, so the procedure illustrated by Example 10–3 is not usually necessary. In addition, the example does not illustrate 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 *VAX/VMS Run-Time Library Routines Reference Manual.*

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.

10.2.5 VAX C Default Parameter-passing Mechanisms

There are default parameter-passing mechanisms established for every data type you can use with VAX C. Table 10–2 shows which 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 10–2: Valid Parameter-passing Mechanisms

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. The following sections describe these routines and the functions they perform.

10.3 VMS Run-Time Library Routines

The VMS Run-Time Library 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 Run-Time Library facility. Table 10–3 lists all the language-independent Run-Time Library facility prefixes and the types of tasks each facility performs.

Table 10–3: Run-Time Library Facilities

Facility Prefix	Types of Tasks Performed
DTK$	DECtalk routines that are used to control DIGITAL's 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.

10.4 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 Run-Time Library 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 10–4 describes these groups.

Table 10–4: 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.

10.5 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 Run-Time Library routine. The following sections outline the procedures for calling non-C routines.

10.5.1 Determining the Type of Call

Before you call an external routine, you must first determine whether the call should be a procedure call or a function call. You should call a routine as a procedure if it does not return a value. You should call a routine as a function if it returns any type of value.

10.5.2 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.

You should include the following information in a routine declaration:

- The name of the external routine
- The data types of all the routine parameters (optional)

The following example shows how to declare an external routine and its arguments.

```
char func_name (int x, char y);
```

10.5.3 Calling the External Routine

Once you have declared 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 10.5.2.

```
ret_status = func_name(1,"a");
```

If you do not want to specify a value for a required parameter, you can pass a null argument by inserting a zero as a placeholder in the argument list.

10.6 Calling VAX C Subprograms from Other Languages

The following sections describe the methods involved in calling VAX C program sections from routines written in other VAX-native languages.

10.6.1 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.

As shown in Example 10–4, a VAX C **extern** variable corresponds to a FORTRAN common block with the same name.

Example 10–4: 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();
}
```

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
```

In Example 10–4, 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 10–5.

Example 10–5: 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 (318)
   RETURN
   END
```

In Example 10–5, the **int** variables declared in the VAX C structure numbers correspond to the FORTRAN INTEGER*4 variables in the COMMON of the same name.

10.6.2 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. Examples 10–6 and 10–7 illustrate the sharing of a program section 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 10–6, VAX C and VAX PL/I use the same variable to manipulate the character string that resides in a program section named XYZ.

Example 10–6: 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** in VAX C. Moreover, VAX PL/I can output aggregates, such as structures and arrays, in fairly simple stream-output statements; consider Example 10–7.

Example 10–7: 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();
}
```

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.

10.6.3 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.

You can set up a psect in a MACRO program to allow data to be shared with a VAX C program, as shown in Example 10–8.

Example 10–8: 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 source code file SETVALUE.MAR contains the following lines:

```
          .entry  set_value,^M<>

          movl     #1,first
          movl     #2,second
          movl     #3,third
          ret

          .psect  example pic,usr,ovr,rel,gbl,shr,-
                  noexe,rd,wrt,novec,long
first:    .blkl
second:   .blkl
third:    .blkl

          .end
```

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.

10.6.4 Calling System Routines

The basic steps for calling system routines are the same as those for calling any external routine. However, when calling system routines, you need to provide some additional information that is discussed in the following sections.

10.6.4.1 System Routine Arguments

All of the 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 10–5 lists all the VMS usages and the VAX C statements you need to implement them.

Table 10–5: VAX C Implementation

VMS Data Type	VAX C Declaration
access_bit_names	User-defined[1]
access_mode	unsigned char
address	int *pointer[2,4]
address_range	int *array [2][2,3,4]
arg_list	User-defined[1]
ast_procedure	Pointer to function.[2]
boolean	unsigned long int
byte_signed	char
byte_unsigned	unsigned char
channel	unsigned short int
char_string	char array[n][3,5]
complex_number	User-defined[1]
cond_value	unsigned long int
context	unsigned long int
date_time	User-defined[1]
device_name	char array[n][3,5]
ef_cluster_name	char array[n][3,5]
ef_number	unsigned long int
exit_handler_block	User-defined[1]
fab	#include fab from text library struct FAB
file_protection	unsigned short int, or User-defined[1]
floating_point	float or double

[1]The 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.

[2]The term *pointer* refers to several declarations involving pointers. Pointers are declared with special syntax and associated with the data type of the object being pointed to. This object is often *user-defined*.

[3]The term *array* denotes the syntax of a VAX C array declaration.

[4]The data type specified can be changed to any valid VAX C data type.

[5]The size of the array must be substituted for n.

Table 10–5 (Cont.): VAX C Implementation

VMS Data Type	VAX C Declaration
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]
page_protection	unsigned long int
procedure	Pointer to function[2]

[1] The 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.

[2] The term *pointer* refers to several declarations involving pointers. Pointers are declared with special syntax and associated with the data type of the object being pointed to. This object is often *user-defined.*

[3] The term *array* denotes the syntax of a VAX C array declaration.

[4] The data type specified can be changed to any valid VAX C data type.

[5] The size of the array must be substituted for n.

Table 10–5 (Cont.): VAX C Implementation

VMS Data Type	VAX C Declaration
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 from text library 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]
word_signed	short int
word_unsigned	unsigned short int

[1] The 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.

[3] The term *array* denotes the syntax of a VAX C array declaration.

[5] The size of the array must be substituted for n.

If a system routine argument is optional, it will be indicated in the format section of the routine description in one of two ways:

- [,optional-argument]
- ,[optional-argument]

If the comma appears outside the brackets (,[optional-argument]), you must pass a zero 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.

10.6.4.2 Symbol Definitions

Many system routines depend on values that are defined in separate symbol definition files. VMS Run-Time Library 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 a simpler 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 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, refer to 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 files, refer to Appendix B, VAX C Compiler Messages.

10.7 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. If a condition value is signaled, then the condition value is signaled to your program instead of being written to a storage location.

Condition values indicating success always appear first in the list of condition values for a particular routine, and success codes always 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:

```
ret_status =
      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.

10.8 Examples of Calling System Routines

This section provides complete examples of calling system routines from VAX C. Example 10–9 shows the three mechanisms for passing arguments to system services and also shows how to test for status return codes. Example 10–10 shows various ways of testing for successful $QIO completion and Example 10–11 illustrates how to use time conversion and set timer routines.

In addition to the examples provided here, the *VAX/VMS Run-Time Library Routines Reference Manual* and the *VAX/VMS System Services Reference Manual* also provide examples for selected routines. Refer to these manuals for help on the use of a specific system routine.

Example 10–9: Passing Arguments to System Services

```
/* GETMSG.C
   This program is an example showing the three mechanisms
   for passing arguments to system services.  It also
   illustrates how to test for specific status return
   codes from a system service call. */

#include stdio
#include descrip
#include ssdef

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 10–10: Determining $QIO Completion

```
/* ASYNCH.C
   This program illustrates various ways to determine
   $QIO completion. It also illustrates 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. */
{
printf("\nNumber of characters output is  %d", write_status->count);
printf("\nI/O completion status is %d", write_status->cond_value);
}
```

Example 10–11: Using Time Routines

```
/*  ALARM.C
    This program illustrates 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=SYS$BINTIM(&offset, delay_int)) &1) != 1)
   LIB$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 timer to expire in 15 seconds */
if (((status=SYS$SETIMR(event_flag, &delay_int,
                        0, timer_id)) &1) != 1)
   LIB$STOP(status);

/* count to 1000000 */
printf("beginning count....\n");
for (i=0; i<=1000000; i++)
   ;
```

(Continued on next page)

Example 10–11 (Cont.): Using Time Routines

```
/* check if 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 cancelled.\n");
                   break;
 case SS$_WASSET : printf("Timer expired before count completed.\n");
                   break;
 default         : LIB$STOP(status);
                   break;
 }
}
```

Chapter 11

VAX C Implementation Notes

This chapter discusses VAX C program sections.

11.1 Program Sections

The following sections describe Program Section Attributes and Program Sections created by VAX C.

11.1.1 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 11–1 lists the attributes that can be applied to program sections.

Table 11–1: Program Section Attributes

Attribute	Meaning
PIC or NOPIC	The program section or the data to which it refers 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	The program section will be concatenated with other program sections with the same name (CON) or will be overlaid 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).

[1] VAX 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.

11.1.2 Program Sections Created by VAX C

When needed, 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 (a string of characters delimited by quotation marks; for example, "space" written in the program, such as the following:

```
char *y = "This is a string *****"

   /*  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; and the $CHAR_STRING_CONSTANTS psect has the same attributes as $DATA. Table 11–2 summarizes the differences in the psects created by VAX C. The first list assigns a number to all the possible combinations of storage class specifiers and modifiers, and the second list presents the psect name and the psect attributes of each of the combinations.

Storage Class Code	Storage Class Keyword Combination
1	[extern]
2	[extern] **const readonly**
3	[extern] **noshare**
4	[extern] **const readonly noshare**
5	**static**
6	**static const readonly**
5	**static noshare**
5	**globaldef**
6	**globaldef const readonly**
5	**globaldef noshare**

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 the first column of the previous table correspond to the numbers in the following table. In the following, 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 11–2: Program Sections for VAX C Variables

Storage Class Code	Program Section Name	Program Attributes
1	*name*	OVR, GBL, SHR, NOEXE, WRT
2	*name*	OVR, GBL, SHR, NOEXE, NOWRT
3	*name*	OVR, GBL, NOSHR, NOEXE, WRT
4	*name*	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

Notice that 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**. Again, the order of the storage class specifier, the storage class modifier, and the data type keyword within a declaration is not significant.

Appendix A
VAX C Definition Modules

This appendix lists the library definition modules contained in the text library named SYS$LIBRARY:VAXCDEF.TLB.

The contents of these modules can be examined in the appropriate definition file. All definition files have the file extension .H and they 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
accdef	Accounting file record definitions
atrdef	File attribute definitions
chfdef	Structure definitions for condition handlers
climsgdef	Command language interpreter error code definitions
ctype	Character type and macro definitions for character classification functions
curses	Curses Screen Management related definitions
dcdef	Device class and type code definitions
descrip	Descriptor structure and constant definitions
devdef	Device characteristics definitions

Table A–1 (Cont.): VAX C Definition Modules

Module	Description
dvidef	$GETDVI system service request code definitions
errno	Error number definitions
errnodef	VAX C error message constants
fab	File access block definitions
fchdef	File characteristics definitions
fibdef	File information block definitions
file	Symbol definitions for **open** function
float[1]	Macro definitions which provide implementation-specific floating-point restrictions.
iodef	I/O function code definitions
jpidef	$GETJPI system service request code definitions
lckdef	Lock manager definitions
lkidef[1]	Lock information data identifier information.
libdef[1]	Definitions of LIB$ return codes
limits[1]	Macro definitions which provide implementation-specific constraints.
lnmdef[1]	Logical name flag definitions
math	Math function definitions
msgdef	System mailbox message type definitions
nam	Name block definitions
nfbdef	DECNET file access definitions
opcdef	OPCOM request code definitions
perror	PERROR function related definitions
pqldef	Process quota code definitions
prcdef[1]	Create process (SYS$CREPRC) system service status flags
prdef	Processor register definitions
prvdef	Privilege mask bit definitions
psldef	Processor status longword definitions

[1] New definition modules.

Table A–1 (Cont.): VAX C Definition Modules

Module	Description
rab	Record access block definitions
rms	All RMS structures and return status value definitions
rmsdef	RMS return status value definitions
secdef	Image section flag bit and match constant definitions
setjmp	State buffer definition for the **setjmp** and **longjmp** functions
sfdef	Stack call frame definitions
signal	Signal value definitions
smgdef	Curses Screen Management interface definitions.
ssdef	System service return status value definitions
stat	STAT and FSTAT function related definitions
stdio	Standard I/O definitions
stsdef	System service status code format definitions
syidef[1]	Definitions for Get System-wide Information (SYS$GETSYI) system service
time	Definitions for the function **localtime**
timeb	Definitions for the function **ftime**
ttdef	Terminal definitions
tt2def	Terminal definitions
types	Type definitions
varargs	Variable argument list access definitions
xab	Extended attribute block definitions
xwdef[1]	System definitions for DECnet DDCMP

[1] New definition modules.

Table A–2 lists each of the modified definition modules and gives a description of the modification:

Table A–2: Modified Definition Modules

Modules	Description of Modification
atrdef	Constant identifier is in uppercase; structure tag is changed from ATTRIB to atrdef.
dcdef	Update incomplete symbol definitions.
dvidef	Update incomplete symbol definitions; constant identifier is in uppercase; structure tag in in lowercase.
fab	Update incomplete symbol definitions.
fchdef	Constant identifiers are in uppercase.
fibdef	Constant identifiers are in uppercase; update obsolete/incomplete symbol definitions.
iodef	Update obsolete/incomplete symbol definitions.
jpidef	Update incomplete symbol definitions.
lckdef	Update obsolete symbol definitions.
msgdef	Update incomplete symbol definitions.
nam	Update obsolete symbol definitions.
opcdef	Update incomplete symbol definitions.
prvdef	Update incomplete symbol definitions.
rmsdef	Update obsolete/incomplete symbol definitions.
smgdef	Update incomplete symbol definitions.
ttdef	Update incomplete symbol definitions.

Appendix B

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 concerning the format of the error messages, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

You can also obtain the compiler diagnostic messages on-line. To do so, type the following:

```
$ HELP  CC  ERROR  mnemonic RETURN
```

To receive a list of all the mnemonics, type the following:

```
$ 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 concerning the /[NO]WARNINGS qualifier, refer to Chapter 1, Developing VAX C Programs at DCL Command Level.

ANACHRONISM, The "****" operator is an obsolete form, and may not be portable.

Informational. You used an old-style assignment operator such as =+ or =*. The message is issued if you specified /STANDARD=PORTABLE on the CC command line.

User Action. For the program to be portable, reverse the order of the operator parts. For example, =+ should be += and =* should be *=. 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.

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, and that structures passed by value occupy as many longwords as are necessary to contain the whole structure.

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.

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 (refer to *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 there exists two **globaldef** definitions 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, you need not take any action; 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 the original CDD item. (VAX C does not support individual bit-fields larger than 32 bits.)

User Action. If the generated declarations are acceptable, you need take no action; 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 (refer to the *VAX C Installation Guide*).

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.

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 usable declarations.

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.

COMPILERR, 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.

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.

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 token-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 zero.

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/or combine the cases, 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.

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.

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, as in

```
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**.

EXTRACOMMA, Extraneous comma in macro parameter list ignored.

Warning. The **#define** macro definition on this line had extra commas that were ignored.

User Action. Make sure that no parameters are omitted in the macro definition.

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 extraneous 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 there exists only one **#module** directive in the source file, and that it is placed before any VAX C source code.

EXTRATEXT, Extraneous text in preprocessor directive ignored.

Informational. Extraneous text appeared in the directive, as in

```
#endif ABC
```

The compiler issued this message if you specified the /STANDARD=PORTABLE qualifier on the CC command line.

User Action. Either remove the extraneous 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 in the indicated line and/or errors 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 does exist. 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 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 being /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 attempted 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. Make sure that no divisors are zero.

IFSYNTAX, 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 token in the source program, and has ignored it. (This may be a syntax error.)

User Action. Make sure that the token and surrounding text is syntactically correct.

INSBEFORE, Inserted **** before ****.

Warning. The compiler attempted to recover from a syntax error by inserting a token into the source.

User Action. Correct the syntax.

INSMATCH, Inserted **** to match **** inserted earlier.

Warning. The compiler attempted to recover from a syntax error by inserting a token to match a previous token in the source code. The previous token may or may not have been inserted by the compiler.

User Action. Make sure that you match all parentheses, brackets, and braces.

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 attempted 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.

INVALIGNSPEC, Invalid alignment specification ignored.

Warning. You specified an alignment option that was not in the allowable range. The compiler ignored the specified option.

User Action. Correct the alignment specification.

INVALINIT, 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 also must 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 storage class modifier, **readonly**.

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.

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, although 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 attempted to cast some object to a structure.

User Action. Correct the expression or cast.

INVDEFNAME, Missing or invalid name in **** preprocessor directive; directive ignored.

Warning. The indicated directive was missing a required name, as in:

```
#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, as in:

```
#dictionary
```

The compiler ignores the entire directive.

User Action. Correct or remove the directive.

INVFIELDSIZE, 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 option was not supported. (The only valid option is the option, main_program.)

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 one 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.

INVLINEFILE, 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.

INVMAINRETVAL, 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, as in:

```
# ABC
```

The directive is 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 attempted to perform an invalid arithmetic operation on a pointer or pointers. The only valid arithmetic operations allowed with pointers are addition and subtraction.

Furthermore, for addition, the only allowable forms are as follows:

```
pointer  +  integer
pointer +=  integer
```

For subtraction, the only allowable forms 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, The "****" storage class is invalid for the declaration of "****".

Warning. You made one of the three following programming errors:

1. You specified a storage class that is invalid in the context in which the declaration appears; for example, specifying **auto** in a declaration located outside of a function.
2. You specified a storage class that is incompatible with another storage class specifier; for example, specifying both **static** and **extern**.
3. You specified a storage class that is incompatible with the data type of the indicated declarator; for example, specifying **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 rather than with the member name.

INVSUBVALUE, Invalid subscript value.

Error. You specified a subscript value which 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;
```

A given tag may only be used 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 "****".

Error. You attempted an invalid variant structure or union declaration such as an array of variants, a pointer to a variant, or a list of variant names.

User Action. Either remove the variant keywords from the declaration or make sure that the keywords are used in a valid structure or union declaration.

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.

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 of 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 tokens into a single token.

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 tokens. Otherwise, check for a missing token 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.

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.

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 zero.

User Action. Correct the expression (but note that its operands must not be floating-point).

NAMETOOLONG, Identifier name exceeds 31 characters; truncated to "****".

Warning. VAX C identifiers are limited to a length of 31 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 inadvertently "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.

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.

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, yet 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 resolved the ambiguity correctly.

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, Comparison of a pointer with a nonzero integer constant or an integer expression may not be portable.

Informational. You compared a pointer to something besides another pointer or the constant 0. This message is issued if you specified /STANDARD=PORTABLE on the CC command line.

User Action. Change the operands or cast them to the same type.

This usage is not portable and is not recommended. The only portable comparison is a comparison between a pointer variable and 0.

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.

Informational. You assigned an integer to a pointer or an address to an integer variable. This message is issued if you specified /STANDARD=PORTABLE on the CC command line.

User Action. Change the operands or cast them to the same type.

This usage is not portable and is not recommended. The only portable assignment is the following:

```
pointer = 0
```

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 the apostrophe is extraneous; 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.

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. (If necessary, your program can continue to output information.)

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.

Warning. A macro definition had more parameters than appeared in its substitution, as in:

```
#define m(a,b,c) a*b
```

User Action. Specify the extra parameter in the substitution or, if it is actually 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, as in:

```
f(a) { int a; }
```

You cannot reference the parameter from within the function.

User Action. If the declaration is simply 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.

PARSTKOVRFLW, 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 attempting 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.

PTRFLOATCVT, Operand of pointer addition or subtraction converted from floating-point to integer.

Warning. You combined a pointer operand with a floating-point value, as in:

```
int i,*ip;
   .
   .
   .
i = ip + 2.;
```

User Action. Make sure that pointers are used only with other pointers or with integers; in the above 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. The compiler issued this message if you specified /STANDARD=PORTABLE on the CC command line.

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 "&".

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 token with a different token. (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. Such reserved names may not be redefined or undefined. They are as follows:

- __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.

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 (refer to Chapter 1, Developing VAX C Programs at DCL Command Level).

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 illustrated syntax error 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.

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.

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 accounted 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 (such 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.

UNDEFIFMAC, "****" is not a currently defined macro; constant zero assumed.

Warning. 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 0.

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, appropriately add the 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 it is used.

UNEXPEND, Unexpected end-of-**** encountered in #define preprocessor directive; directive ignored.

Warning. The end of the **#define** directive or end of the source file was encountered before the definition was complete.

User Action. Check for an incomplete comment within the definition, or 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 zero 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; 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 it unrecognizable. Common errors include misspelled parameters and ambiguous abbreviations.

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 attempted 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 used 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.

Appendix C

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.

C.1 Using VAXLSE with VAX C

The VAX Language Sensitive Editor (VAXLSE) is a powerful and flexible text editor designed specifically for software development. VAXLSE has important features that help you produce syntactically correct code in VAX C.

To invoke VAXLSE, specify the LSEDIT command followed by a file name with a C file type at the DCL prompt. For example:

```
$ LSEDIT USER.C
```

The following sections describe some of the key features of VAXLSE. Section C.1.1 discusses how to enter source code using VAXLSE and Section C.1.2 describes VAXLSE's compiler interface features. Section C.1.3 gives examples of how to generate VAX C source code with VAXLSE.

For more details on advanced features of VAXLSE and VAXSCA, see the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer.*

C.1.1 Entering Source Code Using Tokens and Placeholders

VAXLSE'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.

VAXLSE can be used as a traditional text editor. In addition, you can have the power of using VAXLSE's tokens and placeholders to step through each program construct and supply text for those constructs needing it.

Placeholders are markers in the source code that indicate locations where you can provide program text. These placeholders help you to supply the appropriate syntax in a given context. Generally, you do not need to type placeholders; they are inserted for you by VAXLSE. Placeholders are surrounded by brackets or braces and percent signs.

The types of VAXLSE placeholders are as follows:

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. Generally, 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 issuing 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@}, would 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 issuing the EXPAND command.

VAXLSE provides commands that allow you to manipulate tokens and placeholders. These commands and their default key bindings are as follows:

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 through a screen menu toward the bottom.
↑	Up-arrow	Moves the indicator through a screen menu toward the top.
ENTER RETURN	{ ENTER { RETURN }	Selects a menu option.

To display a list of all the defined tokens provided by VAX C, enter the SHOW TOKEN command.

```
LSE> SHOW TOKEN
```

To display a list of all the defined placeholders provided by VAX C, enter the SHOW PLACEHOLDER command:

```
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 code and to review compilation errors without leaving the editing session, you can use the VAXLSE 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 VAXLSE can use to review compilation errors. The diagnostic information is generated with the /DIAGNOSTICS qualifier that VAXLSE appends to the compilation command.

For example, if you issue the COMPILE command while in the buffer USER.C, the resulting DCL command is as follows:

```
$ CC USER.C/DIAGNOSTICS=USER.DIA
```

VAXLSE supports all of 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 VAXLSE.

The REVIEW command displays any diagnostic messages that result from a compilation. VAXLSE 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 the errors are fixed before you compile your program again.

VAXLSE provides several commands to help you review errors and examine your source code. The following table lists these commands and their default key bindings where applicable.

Command	Key Binding	Function
COMPILE	None	Compiles the contents of the source buffer.
COMPILE/REVIEW	None	Compiles the contents of the source buffer, puts VAXLSE 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 VAXLSE 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.1.3 Examples

This section describes the special features of VAX C available through VAXLSE and provides examples of VAX C code written with VAXLSE.

The following examples show expansions of the more frequently used VAX C tokens and placeholders. The examples are expanded to show the formats and guidelines VAXLSE provides; however, not all of the examples are fully expanded.

The examples show expansions of 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 the line in the code where an action has occurred.

To reproduce the examples, invoke VAXLSE and the VAX C language by using the following syntax:

```
LSEDIT [/qualifier...] filename.C
```

See Section C.1.1 for the commands that manipulate tokens and placeholders.

When the editor is used to create a new VAX C program, the initial string, {@compilation_unit@}, will appear at the top of the screen. Expansion of the initial string will produce the following:

```
->      [@#module@]
        [@module_level_comments@]
        [@include_files@]
        [@macro_definitions@]

        [@preprocessor_line@]...

        [@comment@]...

        [@external_definition@]...

        [@function_definition@]...
```

C.1.3.1 Preprocessor Lines

Erase [@_#module@], [@module_level_comments@], [@include_files@], and [@macro_definitions@]. The cursor will now be positioned on [@preprocessor_line@]. Expand [@preprocessor_line@] to duplicate it and display a menu. Select the option _#include.

```
->      #include
        [@preprocessor_line@]...

        [@comment@]...

        [@external_definition@]...

        [@function_definition@]...
```

When the #include option is selected, 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.

```
->      static [@readonly@]    [@data_type@]    {@init_declarator@}...;
        [@external_definition@]...

        [@function_definition@]...
```

Erase the placeholder [@readonly@], and type the value double over the placeholder [@data_type@].

```
->      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@}.

```
->      static double    number [@= initializer@];
        [@external_definition@]...

        [@function_definition@]...
```

Expand the placeholder [@= initializer@] to display a menu and select the option = {@init_constant_expression@}.

```
->      static double    number = {@init_constant_expression@};
        [@external_definition@]...

        [@function_definition@]...
```

Type the value 30.0 over {@init_constant_expression@}.

```
->      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@}.

```
        [@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.

```
->      [@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@]....

```
->      int    get_string (string, limit)
        [@param_decl@]...
        {
            [@block_decl@]...

            {@statement@}...
        }
```

Expand the placeholder [@param_decl@].

```
        int    get_string (string, limit)
->      [@register@]    [@data_type@]    {@declarator@}...;
        [@param_decl@]...
        {
            [@block_decl@]...

            {@statement@}...
        }
```

Expand the placeholder [@register@] to produce the value register. Type the value char over the placeholder [@data_type@].

```
        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@].

```
        int    get_string (string, limit)
->      register char    string [];
        [@param_decl@]...
        {
            [@block_decl@]...

            {@statement@}...
        }
```

Expand [@param_decl@] again and erase the placeholder [@register@].

```
        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@}.

```
        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@]....

```
        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.

```
->      [@function_level_comments@]
        {@main () OR main function that accept 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 accept arguments from the command line@} to display a menu and select the option main ( ).

```
->      main ()
        {
            [@block_decl@]...

            {@statement@}...
        }
```

Expand the placeholder [@block_decl@] to display a menu and select the option [@data_type@] [@init_declarator@].

```
        main ()
        {
->          [@data_type@]    [@init_declarator@]...;
            [@block_decl@]...

            {@statement@}...
        }
```

Expand option struct will automatically expand to a menu, from which you select the option {@struct struct_decl@}.

```
        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@}...;.

```
        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@].

```
        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@].

```
        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@}....

```
        main ()
        {
            struct
                {
                char    city [20];
->              int    population;
                [@member_decl@]...
                }    [@init_declarator@]...;
            [@block_decl@]...

            {@statement@}...
        }
```

Erase the list placeholder [@member_decl@] and expand [@init_declarator@].

```
        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@]].

```
        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@].

```
        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@}.

```
        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@}...}.

```
        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@}.

```
        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@}.

```
        main ()
        {
            struct
                {
                char    city [20];
                int    population;
                }    data [2] = {
->                                      { "Boston", {@init_constant_expression@} },
                                        [@init_item@]...
                                    },
                        [@init_declarator@]...;
            [@block_decl@]...

            {@statement@}...
        }
```

Type the value 250000 over {@init_constant_expression@}.

```
        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@}.

```
->      [@function_level_comments@]
        {@main () OR main function that accept 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 accept arguments from the command line@} to display a menu and select the option main ( ).

```
->      main ()
        {
            [@block_decl@]...

            {@statement@}...
        }
```

Expand the placeholder {@statement@} to display a menu and select the option if.

```
        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.

```
        main ()
        {
            [@block_decl@]...

->          if ({@binary_expression@})
                {@statement@}
            [@statement@]...
        }
```

Expand {@binary_expression@} to display a menu and select the option {@expression@} {@ { <, > , <=, > =, ==, !=} @} {@expression@}.

```
        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 <.

```
        main ()
        {
            [@block_decl@]...

->          if (count < {@expression@})
                {@statement@}
            [@statement@]...
        }
```

Type the value 10 over {@expression@}.

```
        main ()
        {
            [@block_decl@]...

->          if (count < 10)
                {@statement@}
            [@statement@]...
        }
```

Expand {@statement@} to display a menu and select the option {@expression@};.

```
        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@}.

```
        main ()
        {
            [@block_decl@]...

            if (count < 10)
->              printf ({@expression@}, [@actual_argument@]...);
            [@statement@]...
        }
```

Expand {@expression@} to display a menu and select the option primary again.

```
        main ()
        {
            [@block_decl@]...

            if (count < 10)
->              printf ({@primary@}, [@actual_argument@]...);
            [@statement@]...
        }
```

Expand primary to display a menu and select the option {@string_text@}.

```
        main ()
        {
            [@block_decl@]...

            if (count < 10)
->              printf ("{@string_text@}", [@actual_argument@]...);
            [@statement@]...
        }
```

Type the value less than %d test cases\n over {@string_text@}.

```
        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@]....

```
        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 (VAXSCA) is an interactive source code cross-reference and static analysis tool that works with most VAX programming languages. VAXSCA helps developers keep track of the details of complex, large-scale software systems by displaying source information in response to user queries. VAXSCA uses data generated by the VAX C compiler to supply the requested source information. That information is stored in the VAXSCA library. The data in a VAXSCA library consists of the names of, and information about, all the symbols, modules, and files encountered during a specific compilation of the source.

VAXSCA has both cross-reference and static analysis query features. Cross-referencing supplies information about program symbols and source files. Cross-referencing features include the following:

- Locating names, and occurrences (uses) of these names
- Querying a specified set of names or partial names (wildcarding allowed)
- 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 VAXSCA provide structural information on the interrelation of routines, symbols and files. Static analysis features include the following:

- 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

VAXSCA is fully integrated with VAXLSE to provide extended features. By using VAXSCA with VAXLSE, you can view any portion of an entire system and edit related source files.

Multimodular Development

The cross-referencing and static analysis features of VAXSCA 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/CMS, which is represented in the figure by the CMS Library.

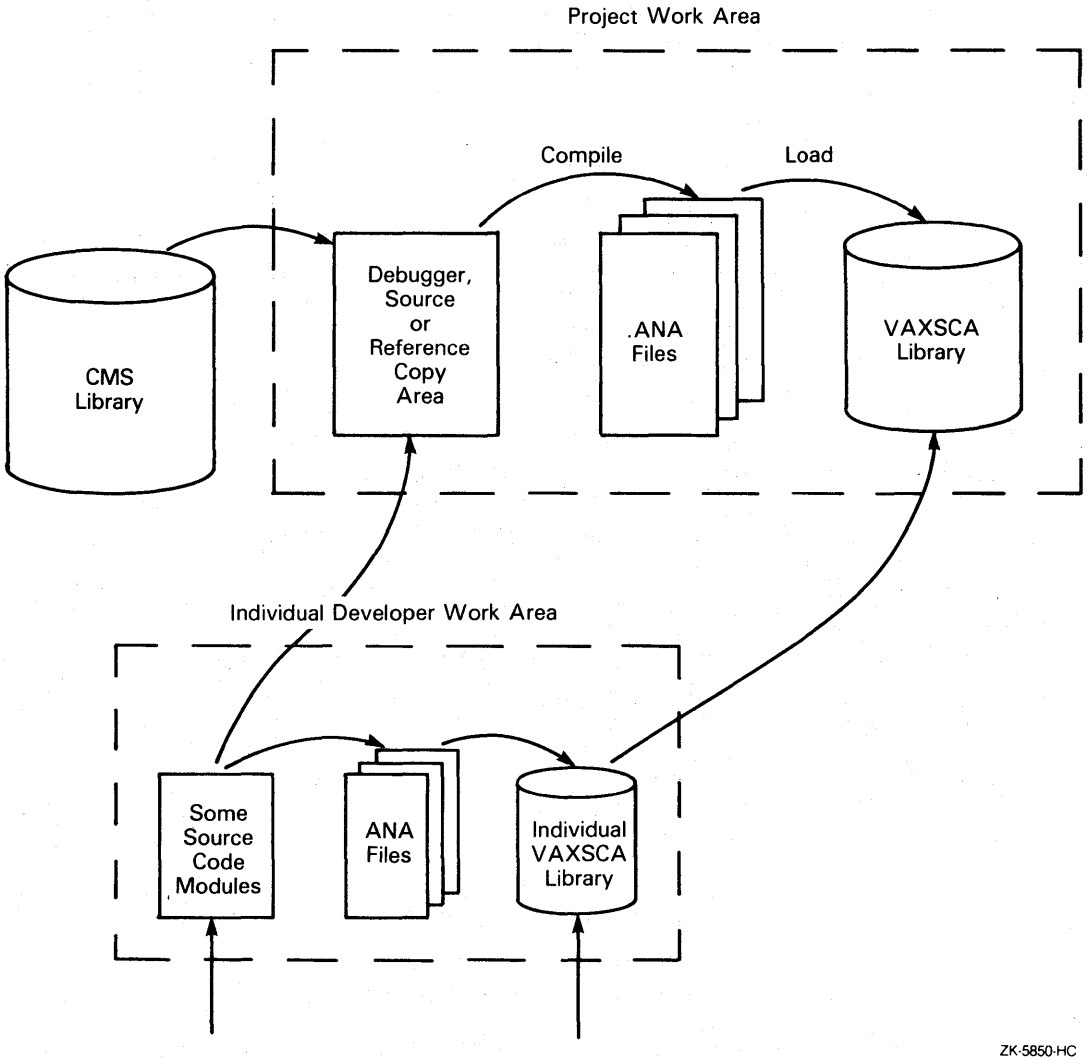
When the team compiles the source code, an /ANALYSIS_DATA qualifier to the COMPILE command instructs the VAX C compiler to generate VAXSCA-required source information (.ANA data files) from the sources. The team then instructs VAXSCA to load the .ANA files into a previously established VAXSCA Library.

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 VAXSCA libraries that contain copies of the module information required to complete assigned tasks.

To make available the module-viewing capabilities of VAXSCA/VAXLSE integration, the project team member must inform VAXLSE of the locations of latest sources, and the related source information. The team member provides pointers to these locations by supplying a search list for VAXLSE. The search list first points to source modules in individual team members' default directories, and then points to the remaining modules in the

Figure C–1: Use of VAXSCA for Multimodular Development



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 of the work can be done with local resources, the team member need not specify the pointers to the project-wide area.

The following sections provide a general overview of VAXSCA and discuss some of the commands that are available to you while using VAXSCA within VAXLSE. For detailed information on VAXSCA and its use with various programming languages, refer to the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer.*

C.2.1 Setting up a VAXSCA Environment

To set up a VAXSCA environment, you must take the following steps:

1. Create a VAXSCA 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 VAXSCA library.

You are then ready to use VAXSCA to conduct source information queries.

C.2.1.1 Creating a VAXSCA Library

To use VAXSCA, you must have a VAXSCA 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 a VAXSCA 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 VAXSCA library.

To initialize a new VAXSCA library you specify the CREATE LIBRARY command. This command has the following form:

```
CREATE LIBRARY [/qualifiers...] directory-spec[,...]
```

For example:

```
$ SCA CREATE LIBRARY [.LIB1]
```

This command intializes and activates library LIB1.

C.2.1.2 Generating the Data Analysis Files

VAXSCA uses detailed source data that is generated by the VAX C compiler. When you specify the /ANALYSIS_DATA qualifier on the CC command, 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.2.1.3 Selecting a VAXSCA Library

To select an existing VAXSCA library to use with your current VAXSCA session, use the SET LIBRARY command. The 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 VAXSCA library selection succeeded.

C.2.1.4 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 a VAXSCA library using the LOAD command. The LOAD 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.2.2 Using VAXSCA for Cross-Referencing

With a VAXSCA library in place, you can ask for symbol or file information by using the VAXSCA command FIND. The FIND command has the following form:

```
FIND [/qualifier...] [name-expression[...]]
```

The name-expression is a 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%
```

You can query VAXSCA library information for the following things that exist within a source program:

Name	A series of characters that uniquely identifies 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, you can use qualifiers on the FIND command, such as the /DECLARATIONS and /REFERENCE qualifiers. For example:

```
LSE> FIND/REFERENCES=CALL BUILD_TABLE
```

This command causes VAXSCA to report only references in the source code where the routine BUILD_TABLE is called.

When you first issue a FIND command within VAXLSE, you initiate a query session. Within this context, the integration of VAXLSE and VAXSCA provides the following commands that can only be used within VAXLSE:

{ NEXT | PREVIOUS } { NAME | ITEM | OCCURRENCE | QUERY | STEP }	Closely-associated commands that let you step through one or more query buffer displays within VAXLSE.
GOTO SOURCE	Displays the source corresponding to the current query item.
GOTO DECLARATION	Positions the cursor on a symbol declaration in one window, and displays the source code that contains the symbol declaration in another window.

Appendix D
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 language features.

D.1 The CC Command

The DIGITAL Command Language (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	Defaults
/[NO]ANALYSIS_DATA[=file-spec]	/NOANALYSIS_DATA
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG[=option]	/DEBUG=TRACEBACK
/[NO]DEFINE[=(definition list)]	/NODEFINE
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS
/G_FLOAT	/NOG_FLOAT
/[NO]INCLUDE_DIRECTORY=(pathname [,...])	/NOINCLUDE_DIRECTORY

/LIBRARY	
/[NO]LIST[=file-spec]	/NOLIST (interactive mode) /LIST (batch mode)
/[NO]MACHINE_CODE[=option]	/NOMACHINE_CODE
/[NO]OBJECT[=file-spec]	/OBJECT
/[NO]OPTIMIZE[=NODISJOINT]	/OPTIMIZE
/[NO]PRECISION={SINGLE,DOUBLE}	/PRECISION=DOUBLE
/SHOW[=(option, . . .)]	/SHOW=(NOBRIEF, NODICTIONARY, NOEXPANSION, NOINCLUDE, NOINTERMEDIATE, NOSTATISTICS, NOSYMBOLS, NOTRANSLATION, SOURCE, TERMINAL)
/[NO]STANDARD=OPTION	/STANDARD
/[NO]UNDEFINE[=(undefine list)]	/NOUNDEFINE
/[NO]WARNINGS[=(option-list)]	/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.

D.2 The LINK Command

The DCL LINK command combines one or more object modules into one image file. If your program contains references to the VAX C Run-Time Library (RTL) functions, read "Specifying Libraries to the Linker" in this section before you link your programs.

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.

Specifying Libraries to the Linker:

If you use any of the VAX C Run-Time Library functions, you must specify libraries for the linker to search in order to resolve references to the functions. You can specify these libraries using the /LIBRARY qualifier on the LINK command line, or you can define the logical name LNK$LIBRARY_n to be the name of an object library. You must define the logicals LNK$LIBRARY_n as the libraries SYS$LIBRARY:VAXCCURSE.OLB, SYS$LIBRARY:VAXCRTLG.OLB, and SYS$LIBRARY:VAXCRTL.OLB. Depending on the needs of your program, you may have to access one, two, or all three libraries. In any case, you must adhere to the following rules for defining libraries for the linker to search:

1. If you do not need to use the Curses Screen Management package of VAX C RTL functions and macros, and you do not use the /G_FLOAT qualifier on the CC command line, you must define the logical as follows:

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

2. If you plan to use the /G_FLOAT qualifier with the CC command line, but do not plan on using Curses, you must define the logicals as follows:

```
$ DEFINE  LNK$LIBRARY SYS$LIBRARY:VAXCRTLG.OLB [RETURN]
$ DEFINE  LNK$LIBRARY_1 SYS$LIBRARY:VAXCRTL.OLB [RETURN]
```

3. If you plan to use the Curses Screen Management package, but do not plan to use the /G_FLOAT qualifier, you must define the logicals as follows:

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

4. Finally, if you plan to use both Curses and the /G_FLOAT qualifier, you must define the three logicals in the following order:

```
$ DEFINE  LNK$LIBRARY SYS$LIBRARY:VAXCCURSE.OLB [RETURN]
$ DEFINE  LNK$LIBRARY_1 SYS$LIBRARY:VAXCRTLG.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 other default libraries before it assumes that the program is in error.

Instead of using the object code of the VAX C RTL functions, you can, as an option, use the VAX C RTL as a shareable image. To use the VAX C RTL as a shareable image, check with your system manager to make sure that the VAX C compiler and software were installed so as to allow access to the shared images. Specifically, check to make sure that the system manager answered YES to step 4 listed in *VAX C Installation Guide*. If that has been done, you can create an options file.

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

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

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

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

You must *not* include the libraries SYS$SHARE:VAXCRTL.EXE and SYS$SHARE:VAXCRTLG.EXE in the same options file.

Once you have created the appropriate options file, OPTIONS_FILE.OPT, you can compile and link your program with the following commands:

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

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
```

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

The operators are listed from highest precedence to lowest. In the binary operator category, operators appear in descending order of precedence, line by line.

Category	Association	Operator
Primary	Left to right	( ) [ ] → .
Unary	Right to left	! ~ ++ −− (*type*) − * & **sizeof**
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 ;

entry[1]
```

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**

1 Reserved for future use.

Function Argument Conversion

Any argument of type:	Is converted to type:
float	**double**
char	**int**
short	**int**
unsigned char	**unsigned int**
unsigned short	**unsigned int**
array	pointer to array
function	pointer to function

D.7 VAX C 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

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 to 7.

Similarly, the form "\xddd" is used 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
```

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

The name of one of the modules in the following list:

Module	Description	Structure Tag
fab	File access block	FAB
rab	Record access block	RAB
nam	Name block	NAM
xaball	Allocation XAB	XABALL
xabdat	Date and time XAB	XABDAT
xabfhc	File header characteristics XAB	XABFHC
xabkey	Indexed file key XAB	XABKEY
xabpro	Protection XAB	XABPRO
xabrdt	Revision date and time XAB	XABRDT
xabsum	Summary XAB	XABSUM
xabtrm	Terminal Control XAB	XABTRM
rmsdef	Completion status codes	—
rms	All RMS modules	All tags

sys$name

The name of the RMS function being called.

pointer

A pointer to an RMS structure which (optionally) has been initialized by the following prototypes:

Prototype	Description
cc$rms_fab	Initializes the file access block (FAB).
cc$rms_rab	Initializes the record access block (RAB).
cc$rms_nam	Initializes the name block (NAM).
cc$rms_xaball	Initializes the allocation XAB (XABALL).
cc$rms_xabdat	Initializes the date and time XAB (XABDAT).
cc$rms_xabfhc	Initializes the file header characteristics XAB (XABFHC).

Prototype	Description
cc$rms_xabkey	Initializes the indexed file key XAB (XABKEY).
cc$rms_xabpro	Initializes the protection XAB (XABPRO).
cc$rms_xabrdt	Initializes the revision date and time XAB (XABRDT).
cc$rms_xabsum	Initializes the summary XAB (XABSUM).
cc$rms_xabtrm	Initializes the terminal control XAB (XABTRM).

error_function
The name of a signal handling function to be called if an error occurs (optional).

success_function
The name of a function to be called if the RMS function is successful (optional).

rms_structure
The type of structure being pointed to by *pointer*.

The RMS functions return an integer status value.

Glossary

additive operator An operator that performs addition (+) or subtraction (−). These operators perform the arithmetic conversion on each of the operands, if necessary. See also *arithmetic conversion rules.*

aggregate A data structure (array, structure, 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 essentially 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 of which 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 *logical operator, unary operator, binary operator,* and *bitwise operators.*

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 means of 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, use of an array or function name (an array or function identifier with no brackets or parentheses) as an argument always 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. For more information concerning the arithmetic conversion rules, refer to Chapter 5, Expressions and Operators.

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 lower level are the binary operators for multiplication (*), division (/), and mod (%). At the next lower 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, of the same type. Elements of an array can have one of the fundamental types or can be structures, unions, or other arrays (to form multidimensional arrays).

assignment expression An expression of the 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 has taken 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 only once.

assignment operator The combination of an arithmetic or bitwise operator with the assignment symbol (=); also, the assignment symbol by itself. See also *assignment expressions*.

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.

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 which make up the body of a **switch** statement are not activated; **auto** or **register** variables declared in the block are not initialized.

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:

```
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 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 of 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 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.

constant expression An expression involving only constants. Constant expressions are evaluated at compile time and may therefore 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**; **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 Run-Time Library 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 Run-Time Library 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 may be written upon, rearranged, moved to new positions on the screen, and deleted 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).

declaration A statement that gives the data type and possibly the storage class of one or more variables.

DEC/Shell An optional VMS software product available under a separate license that is a command language interpreter based on the UNIX V7 Bourne Shell with commands for interactive program development, device and data file manipulation, and interactive and batch execution. DEC/Shell Run-Time Library 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 specifications.*

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.*

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, they should be used in programs as if they had a distinct data type named in the **enum** declaration.

equality operator One of the operators equal to (==) or not equal to (!=). They are analogous to the relational operators, but at the next lower level of precedence.

exponentiation operator The VAX C language does not provide an exponentiation operator.

expression A series of tokens 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 *tokens.*

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 for that of **float** variables, but the precision is 16 decimal digits, as opposed to seven. 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 need not 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.

fundamental type The set of arithmetic data types plus pointers. In general, the fundamental types in VAX C comprise those data types that can be represented naturally on a particular machine; 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. The **globalvalue** specifier can be used 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 31 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.

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.

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. Function names are not VAX C keywords; they may be redefined by the user.

lexical scope The area in which the compiler recognizes an identifier within a given compilation unit. See also *scope*.

lifetime The length of time for which storage for a variable is allocated. See also *program section*.

link libraries The libraries searched by the VMS Linker in order to resolve external references. You can define these libraries at the DIGITAL Command Language (DCL) command line level using the DEFINE command. Define the logical LNK$LIBRARY as the first library to search; define the logical LNK$LIBRARY_1 as the second library to search, and so forth. Depending on the needs of your program, you have to specify certain libraries in a specific order so that your program links properly.

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 See *internal variable*.

logical expression An expression made up of two or more operands separated by a logical operator. Each operand must be of a fundamental type or must be a pointer or other address expression. Operands do not have to be of the same type. Logical expressions always return one or zero (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 (||).

loops A construct which 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 which evaluates the expression before executing the loop body (the **while** statement), one which evaluates the expression after executing the loop body (the **do** statement), and one which executes the loop body a previously specified number of times (the **for** statement).

lvalue The address in memory that is the location of an object whose contents can be assigned or modified. In this manual, the term is used to describe 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 includes 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. The identifier main_program can be used 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, 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 filename 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, 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 token 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.

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, although 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 manual particularly, between UNIX systems and VMS.

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 token replacements; **#undef**, to cancel a previous **#define**; **#include**, for inclusion of 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); 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 which 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 (<=), greater than or equal to (> =). The result (which is of type **int**) is one or zero, 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. You can access the VAX C Run-Time Library 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 which passes control back to the original program. You can access the VAX C Run-Time Library as a shared image; control is passed to the 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 < < E2 is the result of expression E1 (interpreted as a bit pattern) left-shifted by E2 bits. The value of E1 > > 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**, **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 definition module *stdio* 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 specified stdin as the pointer to the file to read from in the **getc** macro, the macro would read from the terminal. To use this definition, include the definition module *stdio* 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 specified stdout as the pointer to the file to write to in the **putc** macro, the macro would write 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, thereby restricting 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. A symbolic constant may be used wherever a literal is valid.

tags Identifiers that represent a declaration of the data types **struct**, **union**, or **enum**. Tags may be used 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 size, in bytes of its operand).

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 these conditions is true:

- The name is used only once.
- It 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. After editing a window, to see the results you must refresh the window on the terminal screen. See also *refresh*.

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line