ULTRIX

Guide to VAX C for ULTRIX

Order Number: AA–ME83B–TE

Guide to VAX C for ULTRIX ™

Order Number: AA-ME83B-TE

June 1990

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

Revision/Update Information: This manual supersedes the *Guide to VAX C for ULTRIX*, (Order No. AA-KK18A-TE).

Operating System and Version: ULTRIX Version 4.0 or higher

digital equipment corporation
maynard, massachusetts

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

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

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

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

The following are trademarks of Digital Equipment Corporation:

ALL–IN–1
DEC
DEC/CMS
DEC/MMS
DECnet
DECmate
DECsystem–10
DECSYSTEM–20
DECUS
DECwriter
DIBOL
EduSystem
IAS
MASSBUS
PDP
PDT
P/OS
Professional
Q–bus
Rainbow
RSTS
RSX
RT
ULTRIX
UNIBUS
VAX
VAXcluster
VMS
VT
Work Processor

digital™

ZK4585

Contents

Porting C Programs

Chapter 1 Program Portability Considerations

Developing VAX C Programs on ULTRIX

VAX C Programming Concepts

Chapter 4 Program Structure

Chapter 5 Statements

Chapter 6 Expressions and Operators

Chapter 7 Data Types and Declarations

Appendix B Diagnostic Messages

Appendix C Transporting VAX C Programs Between VMS and ULTRIX Systems

Appendix D Language Summary

Index

Examples

Figures

Tables

Preface

This guide provides reference information for using VAX C on ULTRIX [TM] systems. It also contains information on how to develop and debug VAX C programs on the ULTRIX operating system running on VAX hardware. VAX C is not intended for use on RISC hardware.

Intended Audience

This guide is intended for experienced and novice programmers who need reference information on VAX C for ULTRIX systems.

Document Structure

This guide has ten chapters and four appendixes as follows:

Chapter 1 describes portability considerations for migrating C source programs between different compilers and the VMS and ULTRIX operating systems.

Chapter 2 explains how to create and compile and link VAX C programs. It also describes the forms of compiler output that you can select.

Chapter 3 discusses the debugging facilities provided by the dbx debugger and how to use the dbx commands.

Chapter 4 explains the structure of VAX C programs, including an introduction to the language, methods of controlling program flow, and the fundamental structures such as function definitions, keywords, blocks, and comments.

Chapter 5 describes the VAX C statements that provide flow control, conditional executions, looping, and interruption.

Chapter 6 discusses the expressions and operators available in VAX C, including unary, binary, conditional, comma, and assignment. Chapter 6 also explains the rules for data-type conversions.

Chapter 7 explains the data types and declarations that VAX C supports.

Chapter 8 describes the storage classes and allocation.

Chapter 9 explains the purposes and appropriate uses of the various VAX C preprocessor directives.

Chapter 10 explains the purposes and appropriate uses of the various VAX C predefined macros and builtin functions.

[TM] ULTRIX is a trademark of Digital Equipment Corporation.

Appendix A describes how to use the lk linker as a separate tool for linking, instead of using the **vcc** command, which both compiles and links programs.

Appendix B lists all the diagnostic messages produced by the **vcc** command program and the VAX C compiler.

Appendix C describes the mechanisms available to assist in transporting C programs between the VMS and ULTRIX operating systems.

Appendix D provides a summary of the **vcc** command and the language elements of VAX C.

Associated Documents

You may find the following documents useful when programming in VAX C. The last two documents are included if you want to transport VAX C programs between the ULTRIX and VMS operating systems.

- *The C Programming Language*[1] — Provides a more intensive tutorial than that found in the beginning of Chapter 4 of this guide.

  VAX C contains additional features and enhancements to the C language as it is defined in *The C Programming Language*. Therefore, use this guide as the reference for a full description of VAX C.

- *ULTRIX Documentation Set* — Provides information about the ULTRIX operating system and its utilities.
- *Guide to VAX C* — Provides tutorial information that describes using VAX C on the VMS operating system.
- *VMS Master Index* — Provides information on the VAX machine architecture in the VMS operating system environment. (This index identifies manuals that cover individual topics about using the VMS operating system.)

Conventions

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.
% cprog RETURN	In interactive examples, the user's response to a prompt is printed in red; system prompts are printed in black.

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

Convention	Meaning
float x; . . . x = 5;	A vertical ellipsis indicates that not all of the text of a program or program output is shown. 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 the dimensions of a multidimensional array in VAX C source code.
sc-specifier ::= **auto** **static** **[extern]** **register**	In syntax definitions, items appearing 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.
auto storage class **fprintf** function	Boldface type identifies language keywords and the names of independently compiled external functions.

Chapter 1

Program Portability Considerations

This chapter describes the portability issues you must consider when migrating existing C source language programs to run on the VAX C compiler for ULTRIX systems (VAX C/ULTRIX). You may not need to migrate C programs. If this is not a concern to you, go to Chapter 2. Read this chapter for information about writing VAX C/ULTRIX programs for improved portability that might be useful in the future.

You will need this chapter if you fall into one of the following two groups:

- You are a user with source programs written for the portable c compiler (pcc) for ULTRIX
- You are a user with source programs written for the VAX C compiler for VMS (VAX C/VMS) Version 3.0 or higher

The first group of users is interested in the differences between the two compilers. The second group of users must learn about the changes imposed by both the compiler differences and the operating system differences. (The compiler differences are relatively minor.)

This chapter also discusses whether the source program, once modified to compile with VAX C on an ULTRIX system, can be recompiled and executed with its original compiler and operating system, if necessary.

This chapter does not describe how to migrate programs. Appendix C describes the most efficient methods for transporting the programs.

NOTE

> To distinguish when VAX C is running on the ULTRIX or the VMS operating system in this chapter, the following convention is adopted: the operating system name is appended after VAX C, as in VAX C/ULTRIX. The remainder of this manual omits the /ULTRIX designation, (unless it is needed for clarification) since the product being described is VAX C on the ULTRIX system.

This chapter presents the information based on the needs of the two specific groups of users. Refer to the sections corresponding to your portability requirements and study the descriptions that follow.

1.1 Differences Between pcc and VAX C/ULTRIX

VAX C is a C compiler available on VAX ULTRIX as vcc. pcc is the default C compiler on VAX ULTRIX. If you are an ULTRIX user and want to compile and link your existing pcc source programs with the VAX C/ULTRIX compiler, note that there are certain behavioral and linguistic differences between these two C compilers. By reading this section, you can decide if it will be easy or difficult to take the source program, once it is modified for VAX C/ULTRIX, and recompile it successfully on pcc.

1.1.1 Behavioral Differences

This section describes the following behavioral differences between the pcc and VAX C compilers:

- Number of passes and methods to perform preprocessing
- Preprocessor behavioral differences
- Optimization capabilities
- Object file formats
- Listing output
- Default options
- Unavailable options
- Unique options
- Compatibility with the lint utility
- Compiler error messages

One area of difference that does not exist concerns the use of system libraries. VAX C/ULTRIX supports all native ULTRIX system libraries, as does the portable C compiler (pcc). Thus, if you are migrating C programs from pcc to VAX C /ULTRIX, you do not need to change your system library calls.

1.1.1.1 Compiler Phases

The pcc compiler is both a multipass and a multiphase compiler. For example, the first phase of pcc invokes the C preprocessor (cpp) to output a temporary file that is used for compiling during the second phase. The VAX C/ULTRIX compiler, does not perform preprocessing in a separate phase. Instead, VAX C/ULTRIX is a single-phase compiler that integrates its preprocessing functions with its other functions in one pass. (This technique speeds up the compiler by eliminating the separate startup time for reloading a new image.)

As a single-phase, single-pass compiler, VAX C/ULTRIX does not produce assembly code as output, either by default or on request. This approach further increases the overall compilation speed. However, since pcc produces assembly code output in response to the **–S** option of the **cc** command, you may be used to generating assembly language code for subsequent editing. You cannot continue this practice when using the VAX C/ULTRIX compiler.

When you specify the **–E** option on the **vcc** command line, VAX C/ULTRIX preprocesses the file and produces source output. There is not a separate preprocessor; this capability is built into the VAX C/ULTRIX compiler. The cpp preprocessor is no longer invoked when the **–E** option is specified.

The **–Em** option invokes the cpp preprocessor, so that output is identical to that generated by the first phase of the pcc compiler. The **–Em** option generates information for the make utility.

Since VAX C is a one-pass compiler, it does not support forward references in either declarations or code. In pcc, an **extern** declaration of an object can be a forward reference to a later-declared static object. However, in VAX C, these attempts generate the warning message that there is a duplicate declaration. VAX C then takes steps to ensure that the two objects are distinct.

1.1.1.2 Preprocessor Behavioral Differences

Since VAX C/ULTRIX and pcc use different code for preprocessing, there are a few anomalies in behavior that may occur under certain conditions.

If the substitution text for a macro identifier also contains the name of that identifier, the VAX C/ULTRIX preprocessor recursively expands the identifier and enters an infinite loop. However, pcc's preprocessor does not do this. If a source program compiled with pcc causes the compiler to enter an infinite loop when migrated to VAX C/ULTRIX, examine the source code for this loop-inducing condition.

1.1.1.3 Optimization Capabilities

The VAX C/ULTRIX compiler employs certain global and many local (also known as peephole) optimizations to generate highly optimized code. As an example of a local optimization, the VAX C/ULTRIX compiler searches for certain combinations of multiple instructions that it can replace with single instructions. As an example of a global optimization, the VAX C/ULTRIX compiler searches for common subexpressions that can be consolidated.

These optimizations occur by default. However, when a VAX C/ULTRIX user compiles a program for debugging with the **–g** option to the **vcc** command, optimization is automatically disabled. Optimization can also be disabled by using **–V nooptimize**. (See Chapter 2.)

1.1.1.4 Object File Formats

In V4.0 of VAX C/ULTRIX, the vcc compiler now generates BSD .o format for its object files. This means that the ld linker can now be used to link object files generated by vcc. Consequently, files can be linked much faster, and standard ULTRIX utilities can now process VAX C/ULTRIX files.

The ld linker is the default linker for VAX C/ULTRIX Version 4.0 files, but ld will not link files produced by a version of vcc prior to Version 4.0, or files with a .obj extension.

The previous object file format can still be generated when you use the **–V lk_object** option on the **vcc** command. This specification causes the **vcc** shell to pass files to the lk linker instead of the default ld linker.

Both the **–V lk_object** and the **–V nolk_object** produce object files with .o extensions. Even though though both object file formats have the same extension, the ld linker will not link files produced with the **–V lk_object** option.

1.1.1.5 Listing Output

The VAX C/ULTRIX compiler can produce a listing that displays the source code, the symbol table, the machine code and cross-reference information, if you specifically request it through the **–v** option of the **vcc** command. (By default, no listing is output.) However, pcc cannot produce a similar listing, either by default or on request.

1.1.1.6 Default Options

There is only one difference in the options that are set by default for the pcc and the VAX C/ULTRIX compilers. With pcc, optimization is off by default; with VAX C/ULTRIX, optimization is on by default.

1.1.1.7 Unavailable Options

VAX C/ULTRIX does not support the following command-line options that are available with pcc:

Option	Meaning
–go	Generates symbol table information for sdb (obsolete).
–R	Makes initialized variables shared and read only (done in assembler).
–S	Generates an assembly language file that can be compiled with the assembler.

1.1.1.8 Unique Options

VAX C/ULTRIX offers some unique options at the command-line level. Among these are the aforementioned **–V standard=portable** option for reviewing portability and the **–v**"*filename*" option for generating listings.

When you invoke the VAX C/ULTRIX compiler, the option **–V standard=portable** is off by default so portability warnings are not automatically generated. However, you can request portability warnings by enabling this option. With pcc, the only way to obtain portability warnings is to invoke the lint utility.

In addition, pcc accepts input from standard input, which **vcc** does not. For more information about these options, see Chapter 2.

1.1.1.9 Compatibility with lint

If you use pcc and are used to using lint to check source programs, you may find that lint reports problems in VAX C source programs that are not real problems when the programs are compiled with VAX C/ULTRIX. For example, lint reports the use of the VAX C language extensions (which are explained in Sections 1.1.2.7 through 1.1.2.13) as problems.

As it compiles, VAX C/ULTRIX performs many of the same checks that lint provides, if you specify **–V standard=portable** on the **vcc** command line. Thus, VAX C/ULTRIX users may prefer not to use lint and to depend on the other methods for checking source programs that are provided by VAX C/ULTRIX. See Chapter 4 for more information about the alternatives to lint for VAX C/ULTRIX users.

1.1.1.10 Compiler Error Messages

The format of the error messages generated by VAX C/ULTRIX and pcc is very similar, but the contents are quite different. See the appropriate compiler documentation for assistance with the meanings of messages.

1.1.2 Language Differences

The language differences that exist between VAX C/ULTRIX and pcc have the following distinct origins which provide a convenient method of classification:

- VAX C includes some features from the draft proposed ANSI standard
- VAX C includes some of its own language extensions
- The compiler designers made different choices for certain similar capabilities or chose not to implement other features

The first two categories of differences show VAX C/ULTRIX to be a superset of the C language as implemented for pcc. The last category highlights a few incompatibilities. This section reviews all the differences in their related groups, since the implications for portability are consistent among the groups.

The primary language differences between pcc and VAX C/ULTRIX stem from the inclusion in VAX C of some additional features that are currently defined in the draft of a proposed ANSI standard for the C language. However, since this is a draft of a proposed standard and is subject to change, Digital reserves the right to change the VAX C language accordingly.

The items in the following list result from the VAX C/ULTRIX incorporation of the proposed ANSI standard features. Sections 1.1.2.1 through 1.1.2.6 describe them in more detail.

- Function prototypes
- Generic **void** pointers
- The **#pragma** preprocessor directive
- Hexadecimal characters in escape sequences
- Vacuous tag declarations
- Additional predefined macros
- The **const** and **volatile** modifiers

The following list summarizes the additional language differences due to the VAX C language extensions. Sections 1.1.2.7 through 1.1.2.13 describe the VAX C language extension differences in more detail.

- The **globaldef**, **globalref**, and **globalvalue** storage class specifiers
- The **noshare** and **readonly** storage class modifiers
- The **variant_struct** and **variant_union** data types
- The **_align** modifier
- The ability to address a constant in a function call
- Multicharacter constants
- The main_program option

There appear to be several differences presented in the two previous lists, but if you want to migrate a program from pcc to VAX C/ULTRIX you should not encounter difficulty, since these differences represent additional VAX C/ULTRIX capabilities. In fact, you may want to take advantage of these differences by recoding segments of existing pcc source programs for recompilation on VAX C/ULTRIX. These differences only represent restrictions to those contemplating running a VAX C/ULTRIX or VAX C/VMS source program on pcc. In that case, carefully rework the source code to remove all use of the features, or the program will fail to compile.

There are a few additional language-related differences that derive neither from VAX C extensions nor the proposed ANSI standard features, but require discussion. This category of differences (described in more detail starting with Section 1.1.2.14), presents the most serious difficulty for porting programs between pcc and VAX C/ULTRIX. These differences include the following:

- Specification of dollar signs ($) in identifiers
- Order of evaluation of subexpressions
- Initialization of external objects
- Compiler-generated global symbols
- The asm pseudo function call
- Variable initialization
- Functions which return a structure value
- Casts as lvalues

In spite of the incompatibility that these differences represent, they are very specific and so narrow in scope that you should rarely encounter them. Nevertheless, if a program incorporates one or more of these features, it may or may not successfully compile on both compilers or it may produce different results when run. Study these differences and then check your C programs for all possible instances, making necessary modifications before trying any program migration.

1.1.2.1 Function Prototypes

VAX C/ULTRIX permits function prototypes as described in Chapter 4, while pcc does not. Therefore, you should not specify function prototypes in your source code if you desire portability between the pcc and VAX C/ULTRIX compilers.

Function prototypes offer many important advantages. With function prototypes, the compiler can verify between definition and invocation whether the types of argument are assignment-compatible or whether different numbers of arguments exist. In this way, you can designate consistent typing of arguments. Some of the semantic checking that the VAX C compiler provides with function prototypes has traditionally been done on ULTRIX systems with the lint utility.

In the presence of a prototype, VAX C/ULTRIX may generate a different argument block based on the argument types specified in the prototype. The compiler can also provide optimizations based on the types of arguments in the argument block. VAX C/ULTRIX always promotes characters and short integers to integers. Since the type **float** passes a single-precision, floating-point argument and not a double-precision, floating-point argument, you must remember that on ULTRIX all math functions expect double-precision arguments to be passed.

Certain include files on VMS systems define functions using function prototypes. Their ULTRIX counterparts do not. Porting a C program from a VMS to an ULTRIX system, which depends on the type conversion implied by a function prototype on a VMS system may produce unexpected results on an ULTRIX system.

1.1.2.2 Generic Pointers

VAX C/ULTRIX permits the use of a generic pointer. The generic pointer is designated by **void** *, as defined in the draft proposed ANSI standard. For example, the following statement shows how you might use the generic pointer in a function prototype:

```
int memcpy (void *destination, void *source, int length);
```

In this case, the function memcpy is an object-copying function that takes three arguments. The first argument specifies the location that will receive the data, the second argument specifies the location of the data to be copied, and the third argument provides the number of bytes to be copied. The data types of the source and destination are not important to the operation of this function, but the VAX C compiler expects the types to be specified, and checks for compliance. To circumvent the compiler's typing requirements, the **void** * generic pointer is used in place of a data type specification for the first two arguments. It specifies that its associated argument is a pointer to data whose type can be arbitrary. Thus, arbitrary data types are successfully copied with this function.

See Chapter 7 for more information about the generic pointer.

This limitation presents no problems in migrating source programs from pcc to VAX C, but is identified here so you remain aware that using generic pointers in VAX C source programs prohibits their compilation by pcc.

1.1.2.3 The #pragma Preprocessor Directive

Another C language feature unique to VAX C/ULTRIX is the **#pragma** preprocessor directive. This directive allows VAX C/ULTRIX users to selectively enable or disable various default compiler behaviors. Source programs that include the **#pragma** directive do not successfully compile with pcc. See Chapter 9 for more information on **#pragma**.

1.1.2.4 Hexadecimal Characters in Escape Sequences

VAX C/ULTRIX allows the specification of hexadecimal characters in escape sequences; pcc does not. For more information, see Chapter 7.

1.1.2.5 Vacuous Tag Declarations

VAX C/ULTRIX allows the use of vacuous tag declarations that eliminate ambiguity in forward references to structure and union tags. For more information about vacuous tag declarations, see Chapter 7. Vacuous tag declarations are not permitted by pcc.

1.1.2.6 Additonal Predefined Macros

VAX C/ULTRIX allows the following macros, which are not recognized by pcc:

- __DATE__
- __TIME__

See Chapter 9 for more information about these predefined macros.

1.1.2.7 Storage-Class Specifiers

The **globaldef**, **globalref**, and **globalvalue** storage class specifiers are VAX C language extensions that are not available with pcc. See Chapter 8 for more information about storage class specifiers.

1.1.2.8 Storage Class Modifiers

The **noshare** and **readonly** storage class modifiers are VAX C language extensions that are not supported by pcc. See Chapter 8 for more information about storage-class modifiers. The **noshare** storage-class specifier is included in VAX C/ULTRIX only for reasons of compatibility with VAX C/VMS; it has no meaning in the ULTRIX implementation of VAX C.

1.1.2.9 The Variant Structure and Union Declarations

The **variant_struct** and **variant_union** declarations are VAX C language extensions that are not supported by pcc. See Chapter 7 for more information about these declarations.

1.1.2.10 The _align Modifier

The **_align** modifier, which allows you to align objects of any of the VAX C data types on a specified storage boundary, is a VAX C language extension that is not supported by pcc. See Chapter 8 for more information about this modifier.

1.1.2.11 The & Operator in Function Calls

Using the & operator with a constant in the argument list of a function call is allowed by VAX C, but is not supported by pcc.

1.1.2.12 Multicharacter Constants

Multicharacter constants are not allowed by pcc, but up to four characters can be specified in a character constant for VAX C/ULTRIX.

1.1.2.13 The main_program Option

The main_program option defines the main entry point in a program the same way that using the name main does. The main_program option provides a way to give the main function a different name. This feature is not supported by pcc.

See Chapter 4 for more information about the main_program option.

1.1.2.14 Specifying Dollar Signs ($) in Identifiers

Use of the dollar sign character ($) in identifiers is accepted by VAX C/ULTRIX. This is permitted by pcc, though the dollar sign character cannot be the first character of a macro name.

1.1.2.15 Order of Evaluation for Subexpressions

The C language does not define a precise order of evaluation for subexpressions found in either argument lists or general expressions. Thus, the pcc and VAX C compilers have each adopted different orders of evaluation. Subexpressions containing side-effect operators (such as ++ and −−) may produce different results with each compiler.

1.1.2.16 Initializing of External Objects

VAX C/ULTRIX will not allow initialization of the declaration of an external object if the declaration contains the **extern** keyword. With pcc, this notation is allowed.

1.1.2.17 Compiler-Generated Global Symbols

The following global symbols are implemented for pcc, but are not implemented with VAX C/ULTRIX:

- edata
- end
- etext

1.1.2.18 The asm Pseudo Function Call

The asm pseudo function call is allowed by pcc, but is not supported by VAX C/ULTRIX. VAX C/ULTRIX provides capabilities similar to asm through built-in functions; these provide access to directly access some VAX instructions from C code. Unlike asm, instead of these functions providing a string which contains an assembler instruction as the parameter, C variables and expressions are the parameters. For more information on these functions, see Chapter 10.

1.1.2.19 Variable Initialization

The following statement is a legal method of initializing the integer foo with pcc:

```
int foo 123;
```

This format is not accepted by VAX C/ULTRIX. The following example shows how the VAX C/ULTRIX implementation requires an equal sign (=) to perform the initialization:

```
int foo = 123;
```

1.1.2.20 Functions Which Return a Structure Value

VAX C/ULTRIX and pcc use different calling conventions to call a function that returns a structure. If you call a function that returns a structure from vcc and that function was compiled with pcc, the call will return unpredictable results. If you call a function that returns a structure from pcc and that function was compiled with vcc, a segmentation fault occurs.

1.1.2.21 Casts as lvalues

VAX C/ULTRIX allows casts to be used as lvalues, while the pcc compiler does not. The following statement is acceptable to vcc, but not pcc:

```
(* new_ptr_type) p = &q
```

1.2 Differences Between VAX C/VMS and VAX C/ULTRIX

If you want to compile and link your existing VAX C source programs with the VAX C/ULTRIX compiler, you must consider the minor behavioral and linguistic differences between the two VAX C compilers. You must learn how to compile, link, and run your source programs on ULTRIX. See Chapter 2 for more information on compiling and linking on ULTRIX.

The major differences discussed in this section can be categorized as follows:

- Language differences
- Include files
- Tool support
- Error message formats
- Structure alignment differences
- Behavioral differences
- Variable names
- The **#module** preprocessor directive

1.2.1 Language Differences

The VAX C/ULTRIX language is a major subset of the VAX C/VMS language, so there are only a few constructs that are not allowed in VAX C/ULTRIX programs.

1.2.1.1 CDD/Plus

If you want to migrate a VAX C program from VMS to ULTRIX, make sure that there are no references to the CDD/Plus data dictionary, which does not exist on ULTRIX systems. Search for all instances where **#dictionary** is specified and remove these references to CDD/Plus.

1.2.1.2 The #include Preprocessor Directive

Additional attention is required wherever the VAX C/VMS source program specifies the following preprocessor directive:

#include *identifier*

The specified identifier is subject to successful macro expansion. While VAX C /VMS allows all of the following possible resulting expansions, VAX C/ULTRIX permits only the specification of file paths in angle brackets (<>) or quotation marks ("):

- **#include** <file-spec>
- **#include** "file-spec"

- **#include** module-name

See Chapter 9 for more information regarding the **#include** preprocessor directive.

1.2.2 Include Files

VAX C/ULTRIX uses the include files provided on ULTRIX systems for pcc. There are differences between the include files on VMS and ULTRIX systems. Therefore, you should closely examine any include files on the system you are using. One significant difference is the **_tolower** and **_toupper** macros. On ULTRIX systems, the definitions add a constant value to their argument (the argument should be tested first with the **islower** and **isupper** macros before being passed to **_tolower** and **_toupper**). Also, the ULTRIX **_tolower** and **_toupper** macros can safely take arguments with side effects such as i++.

On VMS systems, the **_toupper** and **_tolower** macros test to make sure that the argument is in the appropriate range of letters before adding the constant value. The VMS versions of **_tolower** and **_toupper** cannot take arguments with side effects. The ULTRIX versions can.

1.2.3 Tool Support

The ULTRIX system does not provide interface support to two VMS software tools: Source Code Analyzer (SCA) and Language Sensitive Editor (LSE). Consequently, the compiler options **–V diagnostics** and **–V analyze_data** are not supported.

1.2.4 Error Message Formats

The format of the error messages differs between VMS and ULTRIX systems, but the compiler error message content is the same. Appendix B defines the VAX C/ULTRIX error messages.

1.2.5 Structure Alignment Differences

VAX C/VMS does not provide padding of structures and alignment of structure members by default; VAX C/ULTRIX does.

You can use the **#pragma [no]member_alignment** preprocessor directive in your programs to enable or disable the padding of structures and alignment of members at will. See Chapter 9 for more information about this directive.

1.2.6 Behavior Differences

Virtual address 0 is not a valid address on VMS systems. On ULTRIX systems, virtual address 0 is not guaranteed to contain any object. Therefore, a program that dereferences the null pointer (that is, a program that references virtual address 0) causes an access violation error on VMS systems but executes with undefined behavior on ULTRIX systems.

1.2.7 Variable Names

In VAX C/VMS, the maximum length of external variable names is 31. In VAX C/ULTRIX, the maximum length of external variable names is 255.

1.2.8 The #module Preprocessor Directive

The **#module** preprocessor directive is not implemented in VAX C/ULTRIX.

Chapter 2

Developing VAX C Programs for ULTRIX

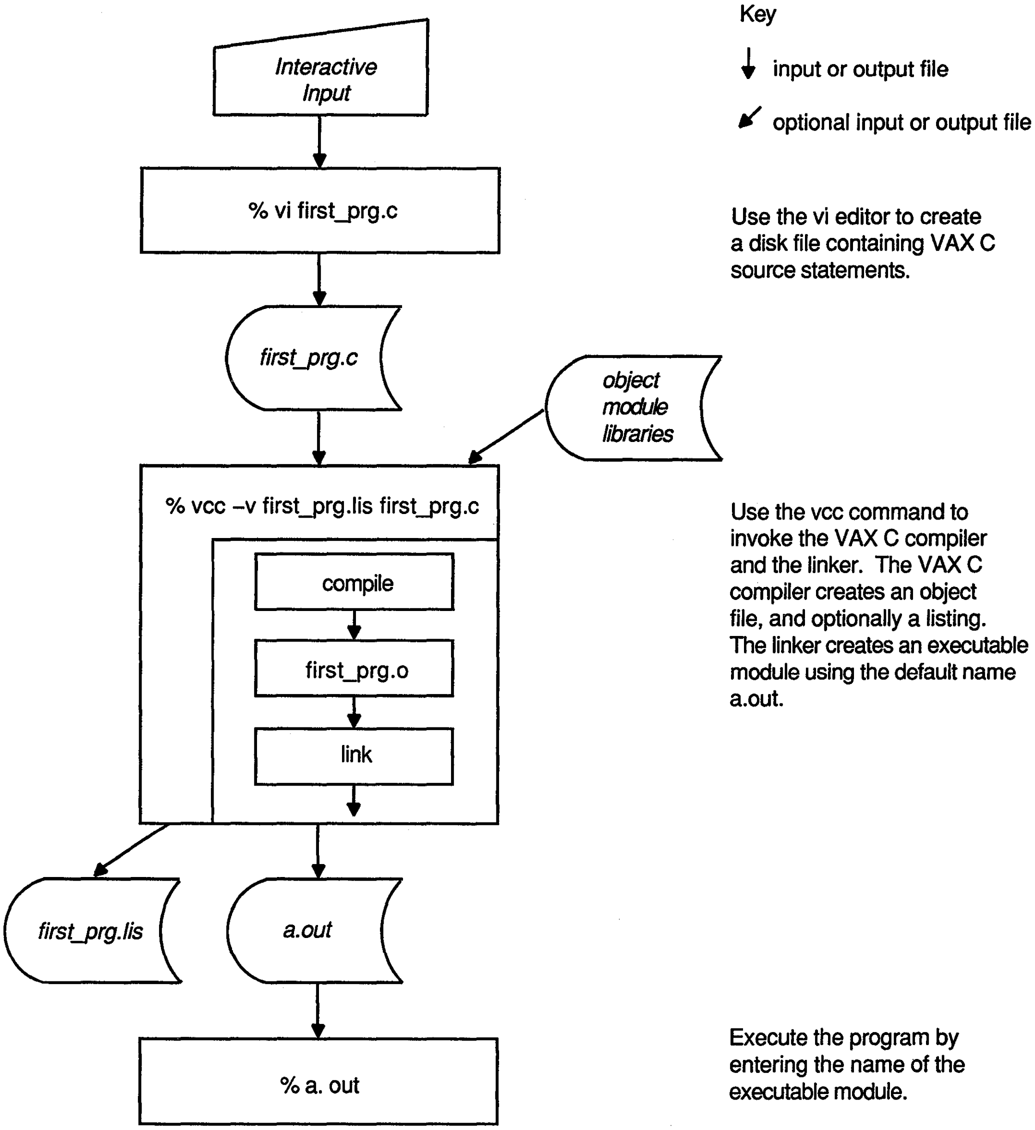
This chapter describes how to develop VAX C source programs and how to use the **vcc** command to compile and link your source programs into object modules and executable images. The following topics are discussed:

- The ULTRIX commands used to create, compile, and link a VAX C program
- The syntax of the vi editor command
- The functions of the compiler and linker
- The syntax of the **vcc** command and its options
- Compiler diagnostic messages and error conditions
- Compiler output listing format

2.1 ULTRIX Commands for Program Development

This section briefly describes the ULTRIX commands you use to create, compile, link, and run a VAX C program. Figure 2–1 shows these commands. For a more detailed description of each command, see the following sections.

Figure 2–1: Commands for VAX C Program Development



Use the vi editor to create a disk file containing VAX C source statements.

Use the vcc command to invoke the VAX C compiler and the linker. The VAX C compiler creates an object file, and optionally a listing. The linker creates an executable module using the default name a.out.

Execute the program by entering the name of the executable module.

ZK–5853–GE

The following example shows each of the commands from Figure 2–1 executed in sequence:

```
% vi first_prg.c
% vcc -v first_prg.lis first_prg.c
% a.out
```

To create a VAX C source program you must invoke a text editor. In the previous example, the vi editor was used to create a program entitled first_prg.c. The c file extension is generally used with C source programs.

After you create a VAX C source program using the vi editor, you can use the **vcc** command to generate an executable module. In most instances, both the VAX C compiler and the linker are invoked and controlled by the **vcc** command. The **vcc** command invokes the compiler, and if the compilation completes without fatal errors and you have not elected to bypass linking, it then invokes the linker. The object files created by the compiler and any linker options and object files specified on the **vcc** command line are passed to the linker.

The **vcc** command creates an optional listing file when you include the **–v** filename option or **–Vlist=**file.lis option on the command line. Section 2.3.3 describes the **vcc** command and its options (including those linker options commonly specified on the **vcc** command).

The executable module, generated by the linker, has the default file name a.out. To execute this program, enter the name of the executable module at the ULTRIX prompt (%).

2.2 Creating a VAX C Program

The vi editor is a screen-oriented display editor that allows you to edit one window of text at a time. A window is a block of text about the size of your terminal screen. When you invoke the vi editor, it copies the file and stores it in an editing buffer. It then displays a window of text from the editing buffer on your screen. Use vi commands to alter the text in the editing buffer. When you finish changing the text, you end the editing session and the vi editor writes the editing buffer to a file.

To invoke the vi editor with the **vi** command, use the following syntax:

vi *filename*

For example, the following command edits, or creates, a file called myprogram.c:

```
% vi myprogram.c
```

After your file is open, you can use the vi commands to enter and edit text.

2.3 Compiling and Linking a VAX C Program

You can use the **vcc** command to compile and link your VAX C programs. This section discusses the **vcc** command and its options along with the functions performed by the VAX C compiler and the linker.

2.3.1 Functions of the Compiler

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

- Verify the correctness of C source statements and to issue informational, warning, and error messages
- Generate machine language instructions from the source statements of the C program
- Group these instructions into an object module that can be processed by the linker

The object file created by the compiler provides the linker with the following information:

- A list of all function, external, and global names declared in the program unit. The linker uses this information when it binds two or more program units together and must resolve references to the same names within the program units.
- A symbol table (if requested with the **–V debug** option or **–g** option on the **vcc** command line as described in Section 2.3.3.4). A symbol table lists the names of all external variables within a module, with definitions of their locations. The table is used in program debugging.

Section 2.3.2 describes the linker.

2.3.2 Functions of the Linker

Use the linker to allocate virtual memory within the executable image, to resolve symbolic references among modules being linked, to assign values to relocatable global symbols, and to perform relocation. The linker's end product is an executable image that you can run on a VAX machine under the ULTRIX system environment.

Normally, you access the linker automatically when you enter the **vcc** command (unless you specify the **–V noobject** option or the **–c** option on the command line).

VAX C/ULTRIX has the capability to use two linkers: the ld linker and the lk linker. The ld linker is the default linker; the lk linker will only be invoked if **–V lkobject** is specified. See the *ULTRIX Reference Pages, Section 1* for more information on the ld linker. See Appendix A for more information on the lk linker.

2.3.3 The vcc Command

The **vcc** command program is your interface to the VAX C compiler. It accepts a list of file names and option switches and causes one or more processors (preprocessor, compiler, assembler, or linker) to process the files.

The **vcc** command has the following form:

vcc [*–options* [*args*]]... *filename*[*.type*] [...*filename*[*.type*]] [*–options* [*args*]]

–options [args]

Indicates either special actions to be performed by the compiler or linker, or special properties of the input/output (I/O) files. See Section 2.3.3.4 for details about command-line options.

filename[.type]

Specifies the source files containing the program units to be compiled. If type is omitted or is not one of the following types, the file is assumed to be an object file and is passed directly to the linker. The following types have special meaning to the **vcc** command and are handled as follows:

.c or .h	Identifies files passed to the C compiler
.s	Identifies files passed to the ULTRIX assembler
.o or .a	Identifies files passed to the linker

You can specify more than one source file. If you specify more than one file, each compiles separately and the resulting object files are linked together to form one executable image.

2.3.3.1 Usage Considerations

In many instances it is enough to specify the name of the VAX C source file on the **vcc** command line without specifying any of the command line options. The VAX C compiler processes the file and passes the resultant object file to the linker along with the appropriate run-time libraries. The linker then produces an executable image with the default file name a.out.

You can select from a variety of processing options and specify files other than VAX C source files. The combination of the processing options and the type of each file determines how the **vcc** command handles the processing. If the options that you specify do not negate a certain level of processing, the **vcc** command examines the type of each file and passes the file to one or more of the following processors: the VAX C compiler, the ULTRIX assembler, or the linker.

You can compile a file for a specific system–SYSTEM_FIVE, BSD, or POSIX–by appending the **–Y** option to the **vcc** command or by setting the PROG_ENV environment variable. The **–Y** option defaults to SYSTEM_FIVE; the environment variable defaults to BSD. A **–Y** option specification overrides an environment variable specification. If neither **–Y** nor PROG_ENV is specified, the default is SYSTEM_FIVE. See Table 2–1 for more information on the **–Y** option. See the *ULTRIX Reference Pages, Section 3* for more information on the PROG_ENV environment variable.

If **–YSYSTEM_FIVE** is specified, the **–DSYSTEM_FIVE** parameter is added to the vaxc command and the **–YSYSTEM_FIVE** parameter is added to the linker call. In addition, the linker parameters **–lc**, **–lcg**, or **–lc_p** are preceded by **–lcV**, **–lcVg**, or **–lcV_p** and the linker parameters **–lm**, **–lmg**, or **–lmp** are changed to **–lmV**, **–lmVg**, or **–lmV_p**. Similarily, if **–YPOSIX** is specified, the **–DPOSIX** parameter is added to the vaxc command, the **–YPOSIZ** parameter is added to the linker call, and the linker parameters **–lc**, **–lcg**, or **–lc_p** are preceded by **–lcP**, **–lcPg**, or **–lcP_p**. Other parameters remain unchanged. If **–YBSD** is specified, the parameter **–YBSD** is added to the linker call.

You can also link routines written and compiled in other languages with VAX C source code. For example, suppose that you have a utility routine written in VAX FORTRAN that you want to call from a VAX C program. The VAX C program is contained in a file named myprog.c. The VAX FORTRAN routine is named utilityx and is contained in a file named utilityx.f. You write a JBL program defining a jacket for calling utilityx, and name it uxjacket.jbl. If you want to link the object files of both the FORTRAN routine and the JBL program to the C program, you can enter the following **vcc** command:

```
% vcc --V lkobject myprog.c utilityx.o uxjacket.o
```

This command processes the files as follows:

- Compiles myprog.c with the VAX C compiler
- Uses the lk linker to link the three routines together into the executable program using the default file name a.out

You can specify standard input as well. VAX C accepts input from standard input with the following syntax, which runs the compile directly:

```
% /usr/lib/vaxc sys%input
```

The output file is named sys$input.o.

If you want to insert compiled programs into a library instead of linking them directly into an executable program, use the **–c** option on the **vcc** command. See Section 2.3.3.4 for more information.

For example, to compile three C source files, a.c, b.c, and c.c, into .o files for insertion into a library, enter the following command:

```
% vcc -c a.c b.c c.c
```

This command compiles the programs and generates the output files a.o, b.o, and c.o.

The following diagram shows how the **vcc** command program processes the various types of files (unless a specified command line option negates some part of the processing):

```
x.s         -> [as]       -> x.o -> [linker] -> a.out

x.c, x.h    -> [vcc -c]   -> x.o -> [linker]-> a.out

x.o, x.a                         -> [linker]-> a.out
```

Files types that are not recognized are treated as object files and passed to the linker. If you specify more than one file type on the command line, the **vcc** command processes each file according to its type.

If the linker produces an a.out file, the **vcc** command program deletes the object files that it created. Object files specified on the **vcc** command line with the **–o** option are retained.

2.3.3.2 Specifying Input Files

Include the full input file specification on the **vcc** command line. If the files are not in your current working directory, you must specify their directory locations.

If you specify multiple files, you must separate the file specifications by spaces. Commas and other special characters are not interpreted as file separators; they are interpreted as part of the file name.

2.3.3.3 Specifying Output Files

The output produced by the compiler includes the following file types:

- An object file when the **–c** option is specified on the command line
- An executable file unless the **–c** option is specified on the command line
- A listing file when the **–v** *filename* option or the **–V list=***filename* option is specified on the command line

You can control the production of these files by specifying the appropriate options on the **vcc** command line. If the files are successfully linked, the **vcc** command program deletes the .o files created by the compiler.

NOTE

> Section 2.3.3.4 describes the command line options in detail.

For VAX C files, the compiler generates an object file (.o file type) by default.

During the early stages of program development, you may find it helpful to use the **–c** option to suppress the production of executable modules until your source program compiles without errors. If you do not specify the **–c** option, the compiler generates executable modules as follows:

- If you specify one source file, one object file is generated and is passed to the linker.
- If you specify multiple source files, separated by spaces, each source file is compiled separately and an object file is generated for each source file. The linker is then invoked to link all files into a single executable file.

To produce an executable file with an explicit file specification, you must use the **–o** option (see Section 2.3.3.4). Otherwise, the object file has the name a.out.

The following examples show two **vcc** commands. Each command is followed by a description of the output files that it produces.

```
% vcc -V list=aaa.lis aaa.c bbb.c ccc.c
```

The VAX C source files (aaa.c, bbb.c, and ccc.c) are compiled as separate files. The object files are then passed to the linker, producing the executable file a.out and the listing file aaa.lis. The **–V** option specifies the listing file name on the command line.

```
% vcc -c circle.c
```

The VAX C source file circle.c is compiled, producing an object file named circle.o. The object file is not passed to the linker because the **–c** option is specified.

2.3.3.4 Options to the vcc Command

The **vcc** command options influence how the compiler processes a file. In many cases, the simplest form of the **vcc** command is sufficient. The options are necessary only when special processing is required.

NOTE

> If you include the **–g** option on the **vcc** command line to prepare a file for use with the dbx debugger, optimization is turned off. Optimizations performed by the compiler can cause unexpected behavior when you are using the dbx symbolic debugger.

Table 2–1 summarizes the **vcc** command options.

Table 2–1: Options Supported by the vcc Command

Option	Description
–B *string*	Finds a substitute compiler,a preprocessor, an assembler, or a linker in the files named by *string*. If *string* is empty, use a standard backup version.
–b	Does not pass the library file –lc to the linker.
–c	Generates an object file with a .o file extension. The linked, executable module is not generated.
–D *name=def*	Assigns the specified value (*def*) to *name*. The preprocessor interprets this option. If a definition value is not specified, the name is set equal to 1.
–E	Runs only the vcc preprocessor and sends the result to standard output. The code is preprocessed and all preprocessor directives, such as include file statements, are resolved. The compiler and the linker are not invoked.
–Em	Runs the cpp preprocessor and produces the makefile dependencies. The file is preprocessed; it is not compiled or linked.
–f	Enables single-precision, floating-point arithmetic. Double-precision, floating point arithmetic is the default selection. Procedure arguments are still promoted to double-precision, floating point format.
–g	Generates additional symbol table information for the dbx debugger.
–I*dir*	Specifies the name of the directory containing the relevant include files. A search for included files whose names do not include a directory specification occurs in the directory of the file, the directory named by the **–I** option, and finally in the directories contained in a standard list.
–K	Generates a full MAP table. This is a linker option. It may be specified on the **vcc** command line or the linker command line.
–l*x*	Specifies a library to include in the link process. The variable *x* is an abbreviation for the library and path name /lib/libx.a in which *x* is a string. If the library is not found, the linker searches for /usr/local/lib /libx.a. A library search starts when the library name is encountered. As a result, the placement of the **–l** within the **vcc** or linker command line is significant.
–Md	Specifies the floating-point type as **D_FLOAT** double-precision, floating-point format. This is the default selection. In addition, the linker receives the **–lc** flag.
–Mg	Specifies the floating-point type as **G_FLOAT** double-precision, floating-point format. In addition, the linker receives the **–lcg** flag. If you want to use the math library, with code generated with the **–Mg** option, you must link in the G_FLOAT version of the library by specifying **–lmg** on the linker or **vcc** command line.
–o	Accepts the specified name as the final output file name. This is a linker option. It may be specified on the **vcc** command line or the linker command line.
–O	Invokes the object code improver. The default selection is to perform object code optimization. See Section 2.3.3.5 for information on how to turn off optimization.

(continued on next page)

Table 2–1 (Cont.): Options Supported by the vcc Command

Option	Description
–p,–pg	Prepares object files for profiling. The **–pg** option also invokes a run-time recording mechanism that produces a gmon.out file. This file contains more extensive statistics.
–t [0pal]	Finds only the designated compiler, preprocessor, assembler, or linker in the files whose names are constructed by a **–B** option. In the absence of a **–B** option, these are found in the standard places.
–U*name*	Makes the specified variable undefined within the program. This option is interpreted by the preprocessor.
–v *filename.lis*	Produces the listing file, complete with a cross-reference table and machine code listing.
–V *option*	Compiles the source code using vendor-specific options. Section 2.3.3.5 lists these options.
–w	Suppresses compiler warning messages. Error messages are displayed, but warning messages are not.
–Y[param]	Compiles a file for one of the following systems: SYSTEM_FIVE, BSD, or POSIX. If a parameter other than SYSTEM_FIVE, BSD, or POSIX is specified, a warning is printed and the **–Y** option is ignored. If no parameter is specified, **–Y** defaults to **–YSYSTEM_FIVE**. If multiple **–Y** options are specified, only the last option takes effect, and no warning message is generated. Section 2.3.3.1 describes how these parameters can be set with the PROG_ENV variable and describes linkage considerations.

2.3.3.5 Specifying Vendor-Specific Options

You can use the **–V** option on the **vcc** command line to specify a vendor-specific option. This option accepts DCL-type VAX C compilation control qualifiers as arguments. (DCL, the Digital Command Language, is offered by Digital on the VMS operating system.)

The syntax for the **–V** option is as follows:

–V *args*

The args are either a quoted string of arguments separated by spaces or a series of arguments separated by commas (with no spaces). The arguments are not case sensitive, and you can abbreviate and specify them in any order. You can use the qualifier names and abbreviations from a DCL CC command line as **–V** option arguments. However, do not use slashes (/) because they conflict with ULTRIX file names.

For example, you can enter the following DCL CC command line in the VMS environment:

```
CC /CROSS_REFERENCE/SHOW=INCLUDE/MACHINE_CODE/LIST FOO.C
```

The following two command lines are the VAX C equivalents to the preceding command line within the ULTRIX operating system environment:

```
% vcc -v foo.l -V show=include foo.c
% vcc -V Cross_reference,show=include,list=foo.lis foo.c
```

The command name, file names, and option switches remain case sensitive.

Table 2–2 lists the DCL type options that you can use as arguments to the **vcc** command's **–V** option. These option arguments are described in detail in the list following the table.

Table 2–2: Arguments to the –V Option of the vcc Command

Argument	Argument Values	Negative Form	Default
cross_reference	—	**nocross_reference**	**nocross_reference**
debug=	**[no]traceback**	**nodebug**	**nosymbols**
	[no]symbols		
	all		**traceback**
	none		
define	*definition-list*	**nodefine**	**nodefine**
g_float	—	**nog_float**	**nog_float**
list[=*file-spec*]	*file-spec*	**nolist**	**nolist**
lkobject	none	**nolkobject**	**nolkobject**
machine_code=	**interspersed**	**nomachine_code**	**nomachine_code**
object[=*file-spec*]	*file-spec*	**noobject**	**object**
optimize=	**[no]inline**	**nooptimize**	**optimize**
show=	**[no]intermediate**	**noshow**	**noshow**
	[no]brief		
	[no]expansion		
	[no]include		
	[no]symbols		
	source		
	terminal		
standard=	**portable**	**nostandard**	**nostandard**
undefine	*undefine-list*	**noundefine**	**noundefine**
warnings=	**nowarnings**	**warnings**	
	noinformationals		

[no]cross_reference

The **cross_reference** argument to the **–V** option specifies that the storage map section of the listing file includes information about the use of symbolic names. The cross-reference contains the numbers of the lines in which the symbols are defined and referenced.

The option argument has the following form:

```
–V [no]cross_reference
```

The **cross_reference** argument is ignored if you do not generate a listing file, and you do not specify **–V show=symbols**.

The default is **nocross_reference**.

See Section 2.5.3 for a description of the listing format that is generated when you specify the **cross_reference** argument.

[no]debug

The **debug** argument to the **–V** option causes the compiler to provide information for use by the dbx symbolic debugger.

The option arguments have the following form:

```
–V debug=all
–V "debug=([no]symbols,[no]traceback)"
–V debug=none
–V nodebug
```

all
Directs the compiler to provide both local symbol definitions and an address correlation table.

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

traceback
Directs the compiler to provide an address correlation table so that the debugger can translate virtual addresses into source program routine names and compiler-generated line numbers.

none
Directs the compiler to provide no debugging information.

If you do not specify the **debug** argument with the **–V** option on the **vcc** command line, the default is **debug=traceback**. Note that **debug** is the equivalent of **debug=all**, and **nodebug** is the equivalent of **debug=none**.

See Chapter 3 and the *Guide to Languages and Programming* for more information on debugging and traceback.

[no]define

The **define** argument to the **–V** option allows you to equate an identifier with a token string or a macro from the command line. Command line definitions override any internal definition statement. Similarly, **undefine** revokes a previous definition.

NOTE

> If the **define** and **undefine** options are both present on the command line, the **define** statement is resolved first.

The option argument has the following form:

–V define=*(identifier[=definition][,...])*

identifier
Specifies the identifier to be defined.

definition
Specifies the value to be associated with the identifier.

The **define** and **undefine** options are functionally equivalent to the **#define** and **#undef** preprocessor directives. The simplest form of the **define** option equates an identifier with the default value, which is 1. For example:

```
-V define=true
```

This command option is equivalent to the following source code statement:

```
#define  true  1
```

You must enclose macro definitions in quotation marks ("). For example:

```
-V define="funct(a)=a+sin(a)"
```

This command option is equivalent to the following source code statement:

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

If you use quotation marks within your definition, a space or a single equal sign is interpreted as a delimiter. Consider the following example:

```
-V define="new_val=51"
```

This command option is equivalent to the following source code statement:

```
#define  new_val  51
```

However, the following command line and source code definitions are equivalent:

```
-V define="new_val =51"
#define  new_val  =51
```

If you do not use quotation marks within your definition, an equal sign is the only recognized delimiter. A space indicates the end of the definition. Consider the following example:

```
-V define=(test1=4,"funct(x)=(a+b)/x")
```

This command option is equivalent to the following source code statements:

```
#define  test1  4
#define  funct(x)   (a+b)/x
```

In the following command line, the definition is not accepted due to the improper use of spaces. In such an instance, an erroneous attempt is made to interpret the information as a file specification instead of part of the definition. This occurs because the space in the statement ends the definition portion of the command line.

```
define= new_val=10
```

As the previous examples show, if you use quotation marks, the compiler interprets either the first space or the first equal sign as a delimiter character. If you do not use quotation marks, the compiler treats the first equal sign as a delimiter. In both instances, any additional equal signs are considered to be part of the value. Thus, special care is required when specifying an equal sign in a value. The following examples show three possible ways to specify an equal sign so that it is treated as part of the value rather than as a delimiter:

```
-V define=(equ==)
-V define=("equa =")
-V define=("equal==")
```

In the first definition, which does not use quotation marks, the first equal sign is a delimiter and the second one is part of the defined value. In the second definition, the space is the delimiter, so the single equal sign is accepted as part of the value. The third example is similar to the first, except that it shows the use of the double equal sign within quotation marks.

[no]g_float

The **g_float** argument to the **-V** option controls how the compiler implements objects of type **double**.

The option argument has the following form:

-V [no]g_float

The default, **nog_float**, causes the compiler to implement double-precision quantities using the VAX D_floating-point data type. Specifying **g_float** causes the compiler to implement double-precision values using the VAX G_floating-point data type.

If your program requires the G_floating-point type form of double-precision data for its correct operation (that is, it uses a range larger than 10^{38}), specify the **g_float** option. See Chapter 7 for more information about using **double** in VAX C.

Routines that pass and receive double-precision quantities should use the same data type as the routines that they pass data to or receive data from. For example, do not pass D_floating point data to a program that uses the G_floating point data type.

CAUTION

> VAX ULTRIX systems support both D_floating and G_floating implementations of type **double**. On different systems, however, the performance of a program can vary widely depending on whether your program is compiled with G_floating or D_floating. The difference exists when a particular system supports one floating-point type in hardware and the other in software. If you want to optimize performance, and if range and accuracy constraints do not dictate one of the two options, you must ensure that the most efficient option is in effect during the compilation.

See Chapter 7 for more information on floating-point data types.

[no]list

The **list** argument to the **-V** option specifies that a source listing file is to be produced.

The option argument has the following form:

```
–V list[=file-spec]
```

You can include a file specification for the listing file. If you do not, the listing file name defaults to the name of the first source file that you specify on your **vcc** command line. This file has the default file extension .lis. The listing file is not automatically printed. You must use the **lpr** command to obtain a line printer copy of the listing file.

Section 2.5.1 discusses the format of a source code listing.

[no]lkobject

The **lkobject** argument to the **-V** specifies that VMS object code format, rather than the default BSD object code format, should be generated for object files. VMS object files should be passed to the lk linker, rather than the default ld linker.

The option argument has the following form:

```
–V [no]lkobject
```

The **vcc** shell, by default, generates BSD .o format for its object files and passes these objects to the ld linker to be linked. If files with a .obj extension appear on the command line, or if **-V lkobject** argument is specified, the files are passed to the lk linker instead.

Object files produced by both **–V lkobject** and **–V nolkobject** have the .o file extension. However, the ld linker will not link files produced with **–V lkobject** or produced by versions of **vcc** prior to Version 4.0. The linker returns an error message stating that the symbol ILLEGAL_LD_OBJECT, USE_LK is undefined.

The default is **nolkobject**.

[no]machine_code

The **machine_code** argument to the **–V** option specifies that the listing file includes a symbolic representation of the object code generated by the compiler. Generated code is represented in a form similar to an assembly listing.

NOTE

> Do not try to assemble and run the object code in the listing file. The code shown in the listing file is similar to VAX MACRO source code; however, the listing file contains constructs not supported by the VAX ULTRIX assembler and is not intended to be used for this purpose.

The option argument has the following form:

```
–V [no]machine_code[=interspersed]
```

If **interspersed** is specified, the listing will consist of lines of source code followed by the corresponding lines of machine code.

If you do not generate a listing file, this option is ignored. The default is **nomachine_code**.

Section 2.5.2 describes the format of a machine code listing.

[no]object

The **object** argument to the **–V** option specifies the name of the object file.

The option argument has the following form:

```
–V noobject
–V object=file-spec
```

The default is **object**, which generates an object file for linking. If you omit the file specification, the object file defaults to the name of the first source file with a .o file extension.

You can use the negative form (**noobject**) to suppress object code generation. This allows you to test for compilation errors in the source program.

[no]optimize[=[no]inline]

The **optimize** argument to the **–V** option specifies that the compiler is to produce optimized code. This optimization takes place within the compiler, not as a separate program. VAX C does not use the object code improver.

The option argument has the following form:

```
–V [no]optimize
```

When you specify **–V optimize=inline**, the compiler performs function inline expansion optimization.

The default is **optimize**. If you include the **–g** or **–V debug** option on your **vcc** command line, optimization is not performed. If you want to debug an optimized program, you must request optimizations with the **–O** or **–V optimize** options to the **vcc** command line.

[no]show

The **show** argument to the **–V** option controls whether or not listing options are selected.

The option argument has the following form:

```
–V show
–V noshow
–V show=all
–V show=[no]brief
–V show=[no]expansion
–V show=[no]include
–V show=[no]intermediate
–V show=none
–V show=[no]source
–V show=[no]symbols
–V show=[no]terminal
```

Use the **list** option with the **show** option to select or cancel any of the options in the previous list. For example, the following command line creates a listing file with the file name mylist.lis. This file includes a listing of the files referenced by the **#include** directive.

```
%vcc programname.c -V list=mylist.lis,show=include
```

all

Specifies that all information is to be included in the listing file.

[no]brief

Generates a listing similar to the one created with the **symbols** option. The only difference between the two is that the **brief** option eliminates any symbols that are not identified in the program, or that do not belong to any union or structure referenced by the program.

The default is **nobrief**.

[no]expansion

Includes the final macro expansions in the listing. When you use this option, the number of substitutions performed on the line is printed next to each line.

The default is **noexpansion**.

[no]include

Includes the modules referenced by the **#include** directives in the listing.

The default is **noinclude**.

[no]intermediate

Includes all intermediate and all final macro expansions in the listing.

The default is **nointermediate**.

none

Generates an empty listing file consisting of header information.

[no]source

Includes source statements in the program listing.

The default is **source**.

[no]statistics

Includes compiler performance statistics in the listing.

The default is **nostatistics**.

[no]symbols
Includes the symbol table in the program listing. The symbol table includes a list of all functions, the size and attributes of each variable, and a program section summary and function definition map.

The default is **nosymbol**.

[no]terminal
Displays compiler messages on the terminal.

The default is **terminal**.

Specifying the **show** argument without any parameters is equal to specifying **show=all**; specifying the **noshow** argument is equal to specifying **show=none**.

standard
standard=noportable
The **standard** argument to the **–V** option specifies that the compiler is to generate informational diagnostics about VAX C language extensions and C code constructs that represent a relaxation of standard C conventions and rules.

The option argument has the following form:

```
–V standard
–V standard=portable
–V standard=noportable
```

You can specify the **standard** option with or without the variable portable; the results are the same. The **standard** option causes the compiler to generate warning messages when it encounters coding practices that are contrary to standard C. This increases portability between VAX C and other implementations of the C language. (This qualifier will not provide warnings about ANSI C features not supported by pcc.)

The default is **standard=noportable**.

If you specify the **nowarnings** argument to the **–V** option, the **standard** argument is ignored.

[no]undefine
The **undefine** argument to the **–V** option allows you to revoke a previous definition. If the **define** and **undefine** options are both present on the command line, the **define** statement is resolved first.

The option argument has the following form:

–V define=*identifier[,identifier ...]*

identifier
Specifies the identifier chosen to have its definition revoked. You can specify a list of identifiers separated by commas.

The **define** and **undefine** options are functionally equivalent to the **#define** and **#undef** preprocessor directives. The **define** option is described earlier in this section.

[no]warnings
The **warnings** argument to the **–V** option causes the compiler to generate informational (I) and warning (W) diagnostic messages in response to informational and warning-level errors.

The option argument has the following form:

```
–V warnings
–V warnings=noinformational
–V warnings=nowarnings
–V nowarnings
```

warnings
The compiler generates informational and warning diagnostic messages. An informational message indicates that a correct C statement may have unexpected results or may contain nonstandard syntax or source forms. A warning message indicates that the compiler detected acceptable, but nonstandard, syntax or performed some corrective action; in either case, unexpected results may occur. To suppress I and W diagnostic messages, specify the negative form of this argument (**nowarnings**).

The default is **warnings**.

warnings=noinformational
The compiler suppresses informational messages. Warning messages are still displayed. The default prints these messages.

warnings=nowarnings
The compiler suppresses all warning messages. Informational messages are still displayed. The default prints these warnings.

nowarnings
The compiler suppresses all messages except for the informational message SUMMARY. The default prints all messages.

Appendix B discusses compiler diagnostic messages.

2.4 Compiler and Linker Diagnostic Messages

Both the compiler and the linker issue error messages that help you to isolate the cause of an error condition. The following sections discuss these messages and error conditions.

2.4.1 Compiler Diagnostic Messages and Error Conditions

One of the functions of the C compiler is to identify syntax errors and violations of language rules in the source program. If the compiler locates any errors, it writes messages to the stderr output file and to any listing file. If you enter the **vcc** command interactively, the messages are displayed on your terminal.
A message from the compiler has the following format:

```
"filename." line nnn: %severity-mnemonic, msg
```

The VAX C compiler replaces the severity code that precedes the message text with one of the following characters:

F	Signifies a fatal condition message
I	Signifies an informational message
W	Signifies a warning message
E	Signifies an error message

S Signifies a success message

Diagnostic messages usually provide enough information for you to determine the cause of an error and correct it.

Each compilation with messages terminates with a summary indicating the number of error, warning, and informational messages generated by the compiler. The summary has the following form:

```
"filename" line nnn: completed with n error(s), n warning(s), and n informational messages.
```

If the compiler creates a listing file, it also writes the messages to the listing file. Messages typically follow the statement causing the error.

Appendix B contains additional information about diagnostic messages, including descriptions of the individual messages.

2.4.2 Linker Diagnostic Messages and Error Conditions

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

Linker messages are descriptive, and you do not normally need additional information to determine the specific error. Some of the more common errors that occur during linking are as follows:

- An object module has compilation errors. This error occurs when you attempt to link a module that had warnings or errors during compilation. Although you can usually link compiled modules for which the compiler generated messages, verify that the modules will produce the output that you expect.
- The modules that are being linked define more than one transfer address. The linker generates a warning if more than one main program has been defined. The image file created by the linker in this case can be run; the entry point to which control is transferred is the first one that the linker found.
- A reference to a symbol name remains unresolved. This error occurs when you omit required module or library names from the **lk** or **ld** command and the linker cannot locate the definition for a specified global symbol reference.

If an error occurs when you link modules, you can often correct it by reentering the command string and specifying the correct routines or libraries.

2.5 Compiler Listings

An output listing produced by the VAX C compiler consists of the following sections:

- A source code section
- A machine code section (optional)
- A storage map section (cross-reference, optional)

Sections 2.5.1 through 2.5.3 describe the compiler listing sections in detail.

2.5.1 Source Code Section

The source code section of a compiler output listing displays the source program as it appears in the input file, with the addition of sequential line numbers generated by the compiler. Example 2–1 shows a sample of a source code section from a compiler output listing.

Example 2–1: Sample Listing of Source Code

```
.MAIN
V1.0

  1          #include "stdio.h"
113          #define STARTNUM 1
114          #define ENDNUM 200
115
116
117          main ()
118          {
119     1        int     Count;
120     1        long    Square;
121     1
122     1        printf ("Table of squares \n\n");
123     1
124     1        for (Count = STARTNUM; Count <= ENDNUM ; Count++){
125     2            Square = (long)Count * (long)Count;
126     2
127     2            printf ("Number: %d Square: %ld\n", Count, Square);
128     2            }
129     1        }
130

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

/usr/lib/vaxc test.c -V list=test.lis
```

Compiler-generated line numbers appear in the left margin.

Compile-time and run-time error messages that contain line numbers refer to these compiler-generated line numbers. (See Appendix B for a summary of error messages.)

2.5.2 Machine Code Section

The machine code section of a compiler output listing provides a symbolic representation of the compiler-generated object code. The representation of the generated code and data is similar to that of an assembly listing. As an option, you can choose to intersperse the machine code with the source code for easier reading.

NOTE

> The machine code is represented in VAX MACRO source code. This code is only for your reference so do not assemble and run it.

The machine code section is optional. To receive a listing file with a machine code section, you can choose one of the following three options:

–v *file.lis programname*
–V list=*file.lis*,machine_code *programname*
–V list=*file.lis*,machine_code=interspersed

Example 2–2 shows a sample of a machine code section from a compiler output listing.

Example 2–2: Sample Listing of Machine Code

```
                       0000 main:
                  0040 0000       .entry main, ^m<r6>
            5E 04 C2   0002       subl2  #4,sp
   56 00000000 EF 9E   0005       movab  $CHAR_STRING_CONSTANTS,r6
               66 DF   000C       pushal (r6)
  00000000* EF 01 FB   000E       calls  #1,printf
            5C 01 D0   0015       movl   #1,ap
                       0018 sym.1:
         52 5C 5C C5   0018       mull3  ap,ap,r2
               52 DD   001C       pushl  r2
            14 A6 DF   0020       pushal 20(r6)
  00000000* EF 03 FB   0023       calls  #3,printf
E6 5C 000000C8 8F F3   002A       aobleq #200,ap,sym.1
            50 01 D0   0032       movl   #1,r0
                  04   0035       ret
                  04   0036       ret
```

The machine code follows the source code listing unless machine_code=interspersed is specified. The object module location of each statement and the machine code instructions are listed. The assembly language code, generated by each line of source text, is shown next to the corresponding machine code instruction.

The following summary outlines the conventions used to represent code and data in machine code listings:

- The VAX MACRO mnemonics represent the generated code.
- R0 through R11 represent the VAX general-purpose registers (0 through 11). Register 12 is an argument pointer that is represented by AP. Register 13 serves as the frame pointer, represented by the mnemonic FP. Register 14 is the stack pointer, represented by SP. Register 15 is the program counter, represented by PC.
- The compiler may generate labels for its own use.
- Signed integer values represent numeric constants.
- The function name plus the hexadecimal offset within that function represent the addresses. Changes from one function to another are indicated by .entry lines.

2.5.3 Storage Map Section

The storage map section of the compiler output listing is printed after each program unit, or function. It summarizes information in the following categories:

- External Declarations Section. The storage map lists all the names declared or defined outside of any function.

- Functions. The storage map lists all the functions in the source program. Along with each name, the following information is listed:
  - The identifier of the name
  - The line on which the name is declared
  - The size of the identifier
  - The storage class of the name
  - The data type of the name
- Function Definition Map. This portion of the storage map lists each function defined in the program and the line number that the function is defined on.

A heading for an information category appears on the listing only when entries are generated for that category.

Cross-reference information is optional. To obtain it, enter the **vcc** command with one of the following options:

–v *filename.lis programname*
–V list,cross_reference *programname*

When you request cross-referencing, the compiler lists the line number where each name is referenced.

Example 2–3 shows a sample storage map section with cross-reference information.

Example 2–3: Sample Storage Map Section

```
.MAIN.                    13-APR-1988 10:35:55    VAX C       V1.0-001     Page 3
V1.0                                                                   test.c (1)

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

External Declarations
---------------------

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

(continued on next page)

Example 2–3 (Cont.): Sample Storage Map Section

```
ctermid             99                     Extern       Function returning
                                                        pointer to char
                                                           - No references
cuserid             99                     Extern       Function returning
                                                        pointer to char
                                                           - No references
freo                96                     Extern       Function returning
                                                        pointer to struct _iobuf
                                                           - No references
ftell               97                     Extern       Function returning
                                                        long int
                                                           - No references
gets                99                     Extern       Function returning
                                                        pointer to char
                                                           - No references
main                117                    Extern def.  Function returning
                                                        long int
                                                           - No references
popen               96                     Extern       Function returning
                                                        pointer to struct _iobuf
                                                           - No references
rewind              98                     Extern       Void function
                                                           - No references
setbuf              98                     Extern       Void function
                                                           - No references
setbuffer           98                     Extern       Void function
                                                           - No references
setlinebuf          98                     Extern       Void function
                                                           - No references
sprintf             102                    Extern       Function returning
                                                        pointer to char
                                                           - No references
temam               100                    Extern       Function returning
                                                        pointer to char
                                                           - No references
tmpnam              100                    Extern       Function returning
                                                        pointer to char
                                                           - No references
_iob                67      60 bytes       Extern       Array [3] of struct
                                                        _iobuf
                                                           - No references
_iobuf              60      20 bytes                    Structure tag
                                                           - Referenced at line 96
 _cnt               61      1 longword                  Member (offset = 0),
                                                        long int
                                                           - No references
 _ptr               62      1 longword                  Member (offset = 4 bytes),
                                                        pointer to char
                                                           - No references
 _base              63      1 longword                  Member (offset = 8 bytes),
                                                        pointer to char

MAIN.                 13-APR-1988 10:35:55   VAX C      V1.0-001       Page 4
V1.0                                                                test.c (1)

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

(continued on next page)

```
 _bufsiz            64      1 longword                   Member (offset = 12 bytes),
                                                         long int
                                                            - No references
 _flag              65      1 word                       Member (offset = 16 bytes),
                                                         short int
                                                            - No references
 _file              66      1 byte                       Member (offset = 18 bytes),
                                                         char
                                                            - No references

Function "main" defined at line 117
-----------------------------------

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

Count               119     1 longword      Register     Long int
                                                            - Referenced at
                                                            lines 124(3),
                                                            125(2), and 127
printf              122                     Extern       Function returning
                                                         long int
                                                            - Referenced at
                                                            lines 122 and 127
Square              120     1 longword      Not Alloc.   Long int
                                                            - Referenced at
                                                            lines 125 and 127

Function Definition Map
-----------------------

 Line   Name
 ----   ----

117     main
Command Line
------------
/usr/lib/vaxc -v test.l -V "cross show=symbol machine" test.c
```

2.6 The lk Linker Image Map

The ld linker cannot generate an image map listing, but the lk linker can. An lk Linker Image Map consists of the following parts:

- An object module synopsis
- A program section synopsis
- A symbol cross-reference
- A symbol value listing
- An image synopsis
- A link-run statistics synopsis

An image map is generated when you specify the **–K** and **–V lkobject** options on the **vcc** command line or the **–K** option on the **lk** command line.

2.6.1 Object Module Synopsis

The Object Module Synopsis shows which object modules (files or library elements) were linked into the program image. Example 2–4 is a sample Object Module Synopsis from the lk Linker Image Map.

Example 2–4: Object Module Synopsis

```
   ❶                                   ❷                          ❸
 a.out             13-APR-1988 13:37     VAX ULTRIX Linker V2.0       Page   1

                            +------------------------+
                            ! Object Module Synopsis !
                            +------------------------+

   ❹       ❺     ❻       ❼         ❽                ❾
Module Name  Ident    Bytes    File         Creation Date       Creator
-----------  -----    -----   -----         -------------       -------
crt0                    124   /lib/crt0.o   24-OCT-1986 16:47  Unknown ULTRIX Compiler
.MAIN.       V1.0       157   test.obj      25-NOV-1986 10:51  VAX C
printf                  80    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
doprnt                2192    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
flsbuf                 296    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
data                   216    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
fclose                 260    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
close                   16    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
getstdiobuf            132    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
fstat                   16    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
isatty                  32    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
gtty                    24    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
exit                    20    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
_exit                    4    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
cerror                  12    /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
$$COMSY,S                8    /lib/libc.a   24-OCT-1986 10:51  VAX ULTRIX Linker
```

Key to Example 2–4:

❶ File name of the program image.

❷ Date and time that the lk linker was run.

❸ Version number of the lk linker.

❹ Name of the object modules.

❺ Module ID, if one is specified.

❻ Size of the module, in bytes (decimal).

❼ File the module was read from.

❽ Date the file was created.

❾ Language processor that created the file, or "unknown ULTRIX compiler" if not known.

2.6.2 Program Section Synopsis

The Program Section Synopsis shows the layout of program sections (psects) and the modules that contributed to them in virtual memory. Example 2–5 is a sample Program Section Synopsis from the lk Linker Image Map.

Example 2–5: Program Section Synopsis

```
a.out            13-APR-1988 13:37        VAX ULTRIX Linker V2.0       Page   2

                          +--------------------------+
                          ! Program Section Synopsis !
                          +--------------------------+

     ❶          ❷          ❸        ❹        ❺                 ❻        ❼
Psect Name  Module Name   Base     End      Length             Align    Attribute
----------  -----------   ----     ---      ------             -----    ----------

ULT$TEXT                00000000 00000F13 00000F14  (  3860.)  LONG 2   PIC,USR,CON
            crt0        00000000 0000004B 0000004C  (    76.)  LONG 2
            printf      0000004C 00000073 00000028  (    40.)  LONG 2
            doprnt      00000074 00000903 00000890  (  2192.)  LONG 2
            flsbuf      00000904 00000A03 00000100  (   256.)  LONG 2
            data        00000A04 00000A77 00000074  (   116.)  LONG 2
            fclose      00000A78 00000B53 000000DC  (   220.)  LONG 2
            close       00000B54 00000B63 00000010  (    16.)  LONG 2
            getstdiobuf 00000B64 00000BBB 00000058  (    88.)  LONG 2
            fstat       00000BBC 00000BCB 00000010  (    16.)  LONG 2
            isatty      00000BCC 00000BEB 00000020  (    32.)  LONG 2
            gtty        00000BEC 00000C03 00000018  (    24.)  LONG 2
            ioctl       00000C04 00000C13 00000010  (    16.)  LONG 2
            malloc      00000C14 00000E4B 00000238  (   568.)  LONG 2
            bcopy       00000E4C 00000EAF 00000064  (   100.)  LONG 2
            sbrk        00000EB0 00000EDF 00000030  (    48.)  LONG 2
            write       00000EE0 00000EEF 00000010  (    16.)  LONG 2
            exit        00000EF0 00000F03 00000014  (    20.)  LONG 2
            _exit       00000F04 00000F07 00000004  (     4.)  LONG 2
            cerror      00000F08 00000F13 0000000C  (    12.)  LONG 2

$CODE$                  00000F14 00000F4A 00000037  (    55.)  LONG 2   PIC,USR,CON,REL
            .MAIN.      00000F14 00000F4A 00000037  (    55.)  LONG 2

$CHAR_STRING_CONSTANTS  00001000 0000102C 0000002D  (    45.)  LONG 2   PIC,USR,CON,REL
            .MAIN.      00001000 0000102C 0000002D  (    45.)  LONG 2

$DATA                   00001030 00001030 00000000  (     0.)  LONG 2   PIC,USR,CON,REL
            .MAIN.      00001030 00001030 00000000  (     0.)  LONG 2

ULT$DATA                00001030 0000119B 0000016C  (   364.)  LONG 2   PIC,USR,CON,REL
            crt0        00001030 0000105F 00000030  (    48.)  LONG 2
            printf      00001060 00001087 00000028  (    40.)  LONG 2
            flsbuf      00001088 000010AF 00000028  (    40.)  LONG 2
            data        000010B0 00001113 00000064  (   100.)  LONG 2
            fclose      00001114 0000113B 00000028  (    40.)  LONG 2
            getstdiobuf 0000113C 00001167 0000002C  (    44.)  LONG 2
            malloc      00001168 00001193 0000002C  (    44.)  LONG 2
            sbrk        00001194 0000119B 00000008  (     8.)  LONG 2
_iob                    000010B4 000010EC 00000039  (    57.)  LONG 2   PIC,USR,CON,REL
            .MAIN.      000010B4 000010EC 00000039  (    57.)  LONG 2

ULT$COMM                0000119C 00001213 00000078  (   120.)  LONG 2   PIC,USR,CON,REL
            malloc      0000119C 0000120B 00000070  (   112.)  LONG 2
            $$COMSYMS   0000120C 00001213 00000008  (     8.)  LONG 2
```

Key to Example 2–5:

❶ Name of the program section.

❷ Names of the modules contributing to the program section.

❸ Starting virtual address (in hexadecimal) of the psect or module.

❹ Ending virtual address (in hexadecimal) of the psect or module.

❺ Length of the psect or module, given in both hexadecimal and decimal.

❻ Alignment specified for the psect. The numeric value is an integer from 0 to 9 that specifies the alignment as a power of two, as shown in the following table:

Value	**Alignment**
0	1 (BYTE)
1	2 (WORD)
2	4 (LONGWORD)
3	8 (QUADWORD)
4	2^4
...	...
9	2^9 (512 bytes)

Alignment on 512-byte boundaries, which is a half page for ULTRIX systems, is the maximum for a psect.

❼ Program section attributes (see Appendix A for more information).

2.6.3 Symbol Cross-Reference

The Symbol Cross-Reference lists symbolic names alphabetically, giving the value of the symbol, the module that defines it, and the modules that refer to it. Example 2–6 is a sample Symbol-Cross Reference table from the lk Linker Image Map.

Example 2–6: Symbol Cross Reference

```
a.out           31-JAN-1987 13:37       VAX ULTRIX Linker V2.0         Page    3
                           +-------------------------+
                           ! Symbol Cross-Reference !
                           +-------------------------+
❶                    ❷          ❸                ❹
 Symbol                 Value          Defined By          Referenced By ...
 ------                 -----          ----------        -----------------
__cleanup               00000A68-R       data              exit
__doprnt                00000174-R       doprnt            printf
__exit                  00000F04-R       _exit             exit
__flsbuf                00000904-R       flsbuf            doprnt
__fwalk                 00000A04-R       data
__getstdiobuf           00000B64-R       getstdiobuf       flsbuf
__iob                   000010B4-R       data              flsbuf        printf
__iobend                0000120C-R       $$COMSYMS         data
__iobstart              000010F0-R       data
_bcopy                  00000E4C-R       bcopy             malloc
_close                  000005BC-R       close             fclose
_edata                  0000119C         <Linker>
 _end                   00001214         <Linker>
_environ                00001034-R       crt0
_errno                  00001210-R       $$COMSYMS         cerror
_etxt                   00000F4B         <Linker>
_exit                   00000EF0-R       exit              crt0
_fclose                 00000AEC-R       fclose            data
_fflush                 00000A78-R       fclose
_free                   00000D04-R       malloc            fclose
_fstat                  00000BC4-R       fstat             getstdiobuf
_gtty                   00000BEC-R       gtty              isatty
_ioctl                  00000C0C-R       ioctl             gtty
_isatty                 00000BCC-R       isatty            flsbuf
_main                   00000F14-R       .MAIN.            crt0
_malloc                 00000C14-R       malloc            getstdiobuf
_moncontrol             00000044-R       crt0
_printf                 0000004C-R       printf            .MAIN.
_realloc                00000D38-R       malloc
_realloc_srchlen        0000116C-R       malloc
_sbrk                   00000EB0-R       sbrk              malloc        flsbuf
_write                  00000EE8-R       write             fclose        close
cerror                  00000F08-R       cerror            _exit         sbrk

curbrk                  00001198-R       sbrk
mcount                  00001038-R       crt0
minbrk                  00001194-R       sbrk
start                   00000000-R       crt0
```

Key to Example 2–6:

❶ Symbol name.

❷ Symbol value, in hexadecimal. See the listing of symbol values for the meaning of the letters suffixed to the values.

❸ Module that defined the symbols. Symbols defined internally by the lk linker display the value <Linker>.

❹ Modules that refer to the symbol (if any).

2.6.4 Symbol Value Listing

The Symbol Value Listing lists the values of the symbols and the symbolic names that have that value. Example 2–7 is a sample Symbol Value Listing from the lk Linker Image Map.

Example 2–7: Symbol Value Listing

```
a.out           13-APR-1988 13:37       VAX ULTRIX Linker V2.0          Page   4
                                  +-------------------+
                                  ! Symbols By Value !
                                  +-------------------+

 ❶                                      ❷
 Value                                  Symbols...
 -----                                  ----------
0000004C                                R-_printf
00000174                                R-__doprnt
00000904                                R-__flsbuf
00000A68                                R-__cleanup
00000AEC                                R-_close
00000B64                                R-__gestdiobuf
00000BC4                                R-_fstat
00000BCC                                R-_isatty
00000BEC                                R-_gtty
00000C0C                                R-_ioctl
00000C14                                R-_malloc
00000D04                                R-_free
00000E4C                                R-_bcopy
00000EB0                                R-_sbrk
00000EE8                                R-_write
00000EF0                                R-_exit
00000F04                                R-__exit
00000F08                                R-_cerror
00000F14                                R-_main
000010B4                                R-_iob
0000120C                                R-_iob_end
00001210                                R-_errno
00001214                                  _end

                             ❸
         Key for special characters above:
                +-------------------+
                ! *  - Undefined    !
                ! R  - Relocatable  !
                ! WK - Weak         !
                +-------------------+
```

Key to Example 2–7:

❶ Hexadecimal value.

❷ Symbols that have the hexadecimal value.

❸ Description of the special characters is as follows:

*	Undefined. (The symbol was not defined anywhere in the link.)

R Relocatable. The definition of the symbol was calculated as an offset from a program section, and thus could change depending on the base virtual address that the lk linker assigns to that psect. If no R appears, the value of the symbol is independent of psect address assignment.

WK Weak. If the lk linker encounters a reference to a symbol that is not defined anywhere, it normally reports this as an error. However, if the symbolic reference is a weak reference, the linker assigns the symbol a value of 0 and does not report an error. If, however, the linker reports any unresolved strong references, it also reports all unresolved weak references.

A weak definition is not included in the _SYMTAB directory of a run-time library. When the linker is searching a library to resolve references (strong or weak), it will not select a module for inclusion on the basis of a weak definition of a symbol. However, if the linker has selected that module for inclusion on the basis of a strong reference to another symbol, it will resolve all references to weak definitions that may be present in that module.

2.6.5 Image Synopsis

The Image Synopsis is a summary of the entire link. Example 2–8 is a sample Image Synopsis from the lk Linker Image Map.

Example 2–8: Image Synopsis

```
a.out                  13-APR-1988 13:37      VAX ULTRIX Linker V2.0    Page    5

                                   +----------------+
                                   ! Image Synopsis !
                                   +----------------+

Virtual memory allocated:                    ❶ 00000000 00001213 00001214
                                                   (4628. bytes, 5. pages)
Text section virtual address limits:         ❷ 00000000 00000FFF 00001000
                                                   (4096. bytes, 4. pages)
Data section virtual address limits:         ❸ 00001000 000013FF 00000400
                                                   (1024. bytes, 1. page)
BSS section virtual address limits:          ❹ 00001400 00001400 00000000
                                                   (0. bytes, 0. pages)
Number of files:                                     5.
Number of modules:                                  21.
Number of program sections:                          8.
Number of global symbols:                           54.
Number of cross references:                         65.
User transfer address:                       ❺ 00000F14
Image type:                                   Demand-loadable (ZMAGIC)
```

Key to Example 2–8:

❶ Total virtual memory allocated to the program.

❷ Limits of the program text (executable) section.

❸ Limits of the initialized data section.

❹ Limits of the uninitialized data (BSS) section.

NOTE

For items 1 through 4, the low and high virtual address of the section and its length is shown in hexadecimal, decimal, and pages.

❺ Address, in hexadecimal, to which control is transferred when the program is run.

2.6.6 Link Run Statistics Synopsis

The Link Run Statistics Synopsis contains statistics and performance indicators for the linker run. Example 2–9 is a sample Link Run Statistic Synopsis from the lk Linker Image Map.

Example 2–9: Link Run Statistics Synopsis

```
                                            +----------------------+
                                            ! Link Run Statistics !
                                            +----------------------+

Performance Indicators                Page Faults  CPU Time        Elapsed Time
----------------------                -----------  --------        ------------
    Command processing:                        15  00:00:00.11     00:00:00.15
    Pass 1:                                    31  00:00:01.13     00:00:01.51
    Allocation/Relocation:                      3  00:00:00.10     00:00:00.20
    Pass 2:                                    10  00:00:00.52     00:00:00.91
    Map data after object module synopsis:      4  00:00:00.51     00:00:00.83
Total run values:                              63  00:00:02.37     00:00:03.60

Using a working set limited to 2097151 pages and 235 pages of data storage
    (including image)

Total number object records read (both passes):           243
    of which 108 were in libraries and 1 were DEBUG data records containing
    77 bytes

Number of modules extracted to resolve undefined symbols:   18

❶ lk /lib/crt0.o test.obj -K /usr/lib/fortrtl.a -lc
```

Key to Example 2–9:

❶ The command line used to invoke the lk linker.

Chapter 3

The dbx Debugger

The dbx debugger is a source-level, symbolic debugger. As a source-level debugger, it allows you to control the execution of individual source lines in a program and to set stops, or breakpoints, at specific source lines. As a symbolic debugger, it also allows you to refer to program locations by their symbolic names.

The dbx debugger supports multiple programming languages, and can evaluate and display values from different languages. In addition, it provides symbol lookup according to the scoping rules of a specific language.

The dbx debugger operates on an executing program and controls the execution according to your specifications. It allows you to follow the execution of your program interactively and enables you to examine or alter the state of your program at any point. It also allows you to access your source file for display and editing purposes.

Before program execution starts, dbx allows you to enter debugger commands (such as setting a breakpoint in your program). You can then enter the debugger's **run** command, which creates a user process and starts program execution. Your program executes until a breakpoint or an exception condition occurs, at which point dbx again gains control and prompts for input. (dbx runs interactively as a separate process; that is, it is not associated with the user process for the executing program.)

NOTE

> You should compile VAX C programs that require debugging without source code optimizations. The VAX C compiler code optimizations will have an unpredictable effect on the debugging environment. When you use the **-g** option on the **vcc** command line optimization is not performed.

Sections in this chapter address the following topics:

- The **dbx** command line used to invoke the dbx debugger
- The conventions observed by dbx
- The effect of compiler optimizations on the dbx debugger
- The commands available within the dbx debugger
- An example of a debugging session

3.1 Invoking the dbx Debugger

The command line that invokes the dbx debugger has the following form:

dbx [–c *file*] [–i] [–I *dir*] [–k] [–r] [*objfile*] [*coredump*]

–c file
Executes the **dbx** commands in the file before reading from standard input.

–i
Forces dbx to act as if the standard input device is a terminal.

–I dir
Adds *dir* to the list of directories that are searched when looking for a source file. Normally, dbx looks for source files in the current directory and in the directory where the file that is being debugged is located. You can set the directory search path with the **use** command. (Section 3.4 contains more information on the **use** command.)

–k
Maps memory addresses. This facility is useful when you are debugging functions within the kernel.

–r
Executes objfile immediately. If it terminates successfully, dbx exits. Otherwise, the reason for termination is reported and dbx does not exit. When the **–r** option is specified and the standard input device is not a terminal, dbx reads from /dev/tty.

If the **–r** option is not specified, dbx issues a prompt and waits for a command.

objfile
Specifies an executable file produced by the **vcc** command. If you do not specify an output file name on the **vcc** command line, the file is given the name a.out by default.

You must specify the **–g** option on the **vcc** command line to produce the symbol information needed by dbx in objfile. The file contains a symbol table that includes the name of all source files translated during the compilation process. You can access these source files while you are using the debugger.

coredump
If the file core exists in the directory, or you specify a coredump file, you can use dbx to examine the state of the program at the time a fault occurs.

NOTE

> Functions compiled without the **–g** option are stepped over by dbx, unless you explicitly set a breakpoint in the function. However, even if you set a breakpoint, you will not be able to fully analyze what is happening in the function because information about the symbols in program components compiled without the **–g** option—except for global symbols—is not available to dbx.

3.2 dbx Conventions

Understanding the conventions discussed in the following sections will assist you when using the dbx debugger.

3.2.1 dbx Initialization Files

You can build an initialization file containing dbx commands that you want to have in effect when you begin debugging sessions. When you enter the **dbx** command, the debugger first searches the current directory for an initialization file. If it fails to find one in the current directory, it searches your home directory. When the debugger finds an initialization file, it executes the dbx commands contained in the file.

In searching for the initialization file, the debugger bases its search on the combination of init and up to the first eight characters of the debugger's name. The debugger normally looks for .dbxinit. If you rename the debugger, it searches for an initialization file consisting of the first eight characters of the new name with the init suffix. If, for example, you rename dbx as ABCDEFGHI, the debugger searches for an initialization file called .ABCDEFGHinit.

3.2.2 Command Line Retention

Each time you enter a **dbx** command, dbx saves that command line until you enter another command. If you want to repeat the execution of the previous **dbx** command line, press the RETURN key.

This feature is useful for repeating a debugging operation (for example, stepping through a program, one instruction at a time, using a **next** or **step** command). You can also use command-line retention to examine consecutive memory locations. The current memory pointer is not automatically updated. The following command sequence allows you to step through memory. This example uses the machine-level debugging commands described in Section 3.4.2.

```
(dbx)  0x100/i
***dbx display***
(dbx)  ./i               <== updates current location pointer
***dbx display***
(dbx)  RETURN
***dbx display***
(dbx)  RETURN
         .
         .
         .
```

3.2.3 Expressions in dbx Commands

You can enter numeric expressions during your dbx debugging sessions. Expressions in dbx commands follow the syntax of the C language. Table 3–1 lists the dbx operators.

Table 3–1: dbx Operators

dbx Operator	Operation
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
= =	Equal to
!=	Not equal to
+	Addition
–	Subtraction
*	Multiplication
/	Division

As in C, you must place array subscripts within square brackets ([ ]). Consider the following example:

```
(dbx)  assign array1[1]=1
```

3.3 Debugging Optimized Programs

The VAX C compiler performs code optimizations by default. Unlike many other optimizing compilers, VAX C does not require additional compile time when optimizing. For many applications, compile and link time is increased when optimization is not used because of the resulting increase in the size of the object program.

Use the **vcc** command line option **–V nooptimize** if you plan to debug VAX C programs with the dbx debugger because code optimizations may affect the debugging environment. To encourage the use of this option, the **–g vcc** command line option turns optimization off.

Debugging optimized code is recommended only under special circumstances. For example, if a problem disappears when optimization is not selected you must try to debug optimized code.

One aid to debugging optimized code is the **–V machine_code=interspersed** option. This option allows you to generate a listing file that shows the compiled code produced for each source line in your program. By referring to a listing of the generated code, you can see exactly how the compiler optimizations affected your code. This helps you to determine the debugging commands needed to isolate the problem.

Another aid is a set of messages that dbx issues when you try to perform a dbx operation on a language construct that has been optimized. In these instances, one of the following optimizations has occurred:

- Optimized variables. If the VAX C compiler determines that a memory location for a variable is not needed for the correct operation of a program, the compiler informs dbx that the variable exists but that no memory is allocated to it. In this case, dbx prints the following message:

  ```
  symbolic information not available for variable variable
  ```

When you receive such a message, you must either find another way to obtain the information you need (perhaps by examining the machine code listing), or you must recompile without specifying the **–V optimize** option on the **vcc** command line.

- Optimized lines. If the VAX C compiler determines that an entire statement is not needed for correct operation of the program, that statement is not represented in the object code. As a result, dbx cannot use such statements to set stops (breakpoints) or tracepoints. If you try to set a stop on an optimized line, dbx prints the following message:

  ```
  no executable code at line line-number
  ```

  If you encounter this situation, you can usually set the stop or trace on an adjacent line that has not been optimized.

  You will also receive the preceding message if you try to set a stop at either a line that contains only data declarations or at a comment line. This occurs whether or not optimization is in effect.

3.4 dbx Commands

Table 3–2 provides a summary of dbx commands. The commands fall into the following functional categories:

- General-purpose commands
- Execution and tracing commands
- Scope and variable handling commands
- Source-file access commands
- Machine-level debugging commands

Sections 3.4.1 and 3.4.2 describe these commands in greater detail.

Table 3–2: dbx Command Summary

Command Category	Command	Description
General-Purpose Commands	**alias**	Establishes an alias for an established dbx command name or lists current aliases.
	help	Displays a summary of dbx commands.
	quit	Terminates dbx processing.
	sh	Passes a command line to the default shell for execution.
	unalias	Removes the **alias** associated with a name.
Execution and Tracing Commands	**call**	Executes the specified function.
	catch	Traps a specified signal condition in the debugger. (In this case, your program will not receive the signal unless you enter a **cont** command.)

(continued on next page)

Table 3–2 (Cont.): dbx Command Summary

Command Category	Command	Description
	cont	Continues execution from the point of suspension. (The **cont** command is also used to continue signal processing.)
	delete	Removes specified traces and stops.
	ignore	Stops trapping a specified signal condition. (In this case, the debugger will automatically pass the signal to your program.)
	next	Executes the current source statement—executing any called subprograms—and stops at the next source line.
	rerun	Reruns a program using the arguments specified on a previous **run** command.
	return	Stops dbx processing at the next executable instruction following a return from the current subprogram or from a specified subprogram.
	run	Begins execution of a program.
	source	Executes dbx commands contained in a given file.
	status	Displays the traces and stops that are in effect.
	step	Executes one source line—stopping at the first line of a called subprogram.
	stop	Suspends execution when a line is reached, when a function is called, or when a condition is met.
	trace	Traces execution of a line, traces calls to a function, or traces changes to a variable. Also, displays the results of specified expressions when a given line is reached.
Scope and Variable Handling Commands	**assign**	Assigns the value of an expression to a variable.
	down	Changes the current scope to a lower stack count level.
	dump	Displays the names and values of all active variables.
	func	Displays or changes the current scope.
	print	Displays the value of an expression.
	set	Assigns values to debugger variables.
	unset	Deletes the debugger variable associated with the name.
	up	Changes the current scope to a higher stack count level.
	whatis	Displays the declaration of a name.
	where	Displays the currently active functions.
	whereis	Displays the full qualification of all occurrences of a symbol.
	which	Displays, in the current scope, the full qualification of a symbol.

(continued on next page)

Table 3–2 (Cont.): dbx Command Summary

Command Category	Command	Description
Source-File Access Commands	**edit**	Begins an editing session on a specified file using the default editor (vi).
	file	Displays or changes the current source file.
	list	Displays a range of source lines or a specified function.
	use	Establishes a directory search path.
	/ ?	Searches forward or backward in the current source file for the specified pattern.
Machine-Level Debugging Commands	*address*	Displays the contents of memory locations at the specified address (or within the specified address range).
	nexti	Executes the current machine instruction and stops at the next machine instruction.
	stepi	Executes the current machine instruction, and stops at the first machine instruction of a called function.
	stopi	Suspends execution when an address is reached, a condition is met, or a function is called.
	tracei	Traces execution of an address.

3.4.1 Source-Level Debugging Commands

This section provides expanded descriptions of the dbx commands. Section 3.4.2 describes the machine-level commands.

alias *newcommandname oldcommandname* alias *commandname*
alias *commandname "string"*
alias *commandname(parameter) "string"*
alias
Responds to newcommandname as though it were oldcommandname. You can use either name to enter the command.

If you enter the **alias** command with a name, a string, and an optional parameter, **alias** executes the string each time you enter the name. For example, to define an alias called b that sets a stop at a particular line, issue the following dbx command:

```
dbx  alias b(x) "stop at x"
```

When you enter the command b(12), dbx stops execution at line 12.

If you enter only one command name, dbx displays the aliases for that name.

If you enter the **alias** command with no arguments, dbx displays all the aliases that you have established.

assign *variable = expression*
Assigns the value of the expression to the variable. The value and the variable must be of the same data type.

You cannot assign values to registers.

call *function(parameters)*
Executes the named function. The dbx debugger passes all parameters by reference; that is, it passes the address of the parameter arguments.

catch *number*
catch *name*
Traps the signal, specified by its name or number, in dbx rather than passing it to the executing program. The **catch** command remains in effect until it is terminated by the **ignore** command.

This command is useful when you are debugging programs that handle signals, such as interrupts. (See the *Guide to Languages and Programming* for more information about signals and signal handling.) By default, dbx traps all signals except SIGCONT, SIGCHILD, SIGALRM, and SIGKILL. The dbx debugger does not keep track of a signal number after it traps it. If dbx traps a signal and you want to pass it to a signal handler routine, you must enter a **cont** command specifying the appropriate signal name or number. (This section describes the **cont** and **ignore** commands.)

cont [*number*]
cont [*name*]
Continues execution from the point at which the process stopped. Execution cannot be continued if the process has finished (that is, it has called the standard exit procedure) or if it has not started executing. dbx does not allow the process to exit so you cannot examine the program state after execution is completed.

The number is an integer (1 through 32) that represents a signal. When you specify the number or name of a signal, you instruct the program that receives that signal to continue processing as if the signal was received. (See the description of the **catch** command that appears in this section.) For example, the following command specifies that a program that receives signal 4 is to continue and that control is to be passed to the appropriate signal handler:

```
(dbx)  cont 4
```

delete *commandnumber, ..., commandnumber*
delete *
Removes the traces or stops corresponding to the given command numbers. (The numbers associated with traces and stops are displayed by the **status** command.)

The **delete** * command removes all existing breakpoints and tracepoints.

down [*count*]
Moves the current scope, which is used for resolving names, down the stack by the number of levels specified in count. The default count is 1. See the **up** and **func** commands for more information.

dump [> *filename*]
Displays the names and values of all active variables in the specified procedure or the current procedure. If you use a period (.) in place of the function name, all active variables are dumped. Active variables are variables in common blocks or in functions that are active. See the **print** command for a list of the supported data types.

edit [*filename*]
edit *functionname*
Invokes the default editor (vi) and accesses the file filename or, if none is specified, the current source file. If you specify a function name, the editor accesses the file containing the specified function.

file [*filename*]

Changes the current source file to filename. If you omit the filename, dbx displays the name of the current source file.

func [*function*]

Changes the current function. If you omit the function, dbx displays the name of the current function. Changing the current function implicitly changes the current source file to the one that contains the function; it also changes the current scope used for name resolution.

The **func** command is similar to the **up** and **down** commands. The major difference is that **up** and **down** change scope based on stack frame counts and **func** changes scope based on the function name that you specify. You can use **func** to change the scope (the current function) to an inactive function (that is, a function not on the stack). However, the **up** and **down** functions are only effective with active functions.

help

Displays a summary of dbx commands.

ignore *number*
ignore *name*

Stops trapping the signal, specified by its name or number and passes it directly to the user command program **ignore**. The **ignore** command disables the **catch** command. (A description of the **catch** command appears in this section.)

The **ignore** command is useful when you are debugging programs that handle signals, such as interrupts. By default, dbx traps all signals except SIGCONT, SIGCHILD, SIGALRM, and SIGKILL. The dbx debugger does not keep track of a signal number after it traps it. If dbx traps a signal and you want to pass it to a signal handler routine, you must issue a **cont** command specifying the appropriate signal number. (A description of the **cont** command appears earlier in this section.)

You can disable trapping for a particular signal using the **ignore** command, as follows:

```
(dbx) ignore 4
```

Signal number 4 (SIGILL) applies to interrupts caused by illegal instructions. If an illegal instruction occurs after you issue the preceding **ignore** command, dbx ignores the signal and gives the program a chance to handle the signal. (See the *Guide to Languages and Programming* for additional information about signals and signal handling.)

list [*linenumber*[,*linenumber*]]
list *function*

Lists the lines in the current source file from the first line number specified to the second line number specified, inclusive. If you omit line numbers the next ten lines are displayed. If you specify the name of a subprogram, ten lines (five above and five below the first statement in the subprogram) are displayed.

The second form of the **list** command, **list** *function*, has the same effect as the **file** command; that is, it changes the current source file.

next

Executes up to the next source line. The **next** command is different from the **step** command. If the current line contains a call to a function and you enter the **next** command, execution continues until the next source line; that is, execution does not stop within the called function. In contrast, if you enter the **step** command execution stops at the first line of the called function.

print *expression*[,*expression* ...]

Displays the values of the specified expressions. Array expressions are always subscripted by brackets ([ ]). You can reference variables that have the same identifier as one within the current scope, as functionname.variable. You can use the backslash operator (\ ) in the construct expression\typename to display the results of an expression in the format of a given type.

The **print** and **dump** commands display variables with the following data types:

Data Type	Size
int **long** **long int**	32 bits
unsigned **unsigned int**	32 bits
short **short int**	16 bits
unsigned short	16 bits
char **unsigned char**	8 bits
float	32 bits
double	64 bits

quit

Terminates the debugging session.

rerun [*args*][< *filename*] [> *filename*]

Starts executing the program specified with filename by passing args as command-line arguments. If, for example, argc and argv are arguments expected by a C program, you can include them on the **rerun** command line. Use left and right angle brackets (<>) to redirect input or output in the usual manner.

If you specify arguments on the **rerun** command, dbx appends them to the original argument list specified by the **run** command. If you omit arguments from the **rerun** command, dbx passes the previous argument list to the program. Otherwise, the **rerun** command is identical to the **run** command.

If the object file associated with filename has been written since the last time the symbolic information was read in, dbx reads in the new information.

return [*function*]

Continues until a return to function is executed, or until the current subroutine or function returns if function is omitted.

run [*args*] [< *filename*] [> *filename*]

Starts executing the program specified with filename by passing args as command line arguments. If, for example, argc and argv are arguments expected by a C program, you can include them on the **run** command line. Use left and right angle brackets (<>) to redirect input or output in the usual manner.

If you specify arguments on the **rerun** command, they are appended to the original argument list specified by the **run** command. If you omit arguments from the **rerun** command, the previous argument list is passed to the program. Otherwise, the **rerun** command is identical to the **run** command.

If the object file associated with filename has been written since the last time the symbolic information was read in, dbx reads in the new information.

set *variable = expression*

Assigns values to debugger variables. The names of these variables cannot conflict with variable names in the program being debugged. The following variables have a special meaning within dbx:

$frame	You can set this variable to an address. The dbx debugger uses the stack frame pointed to by the address to perform stack traces and access local variables.
$hexchars **$hexints** **$hexoffsets** **$hexstrings**	When one of the variables is set, dbx prints out the hexadecimal value for characters, integers, offsets, or strings.
$listwindow	You can set this variable to a numeric value that will be used by the **list** command. The value specifies the number of lines to list around a function, or the number of lines to display when the **list** command is entered without a parameter. If you use the **set** command to set this variable without specifying a numeric value, dbx uses the default value of 10.
$mapaddrs	When you set this variable, dbx starts mapping addresses and continues to map addresses until you revoke the variable setting.
$unsafecall **$unsafeassign**	When you set these variables, strict type checking is omitted within specific situations. The **$unsafecall** variable omits strict type checking during subroutine and function calls. The **$unsafeassign** variable omits strict type checking for the two sides of an assignment statement.

sh *commandline*

Passes the command line to the shell for execution. The mechanism used to return to dbx varies according to the shell being used. For example, for the Bourne shell, you issue CTRL/D. Your shell is determined when you log in.

source *filename*

Executes the dbx commands contained in the specified file.

status [> *filename*]

Displays the currently active **trace** and **stop** commands. You can ignore any gaps in the sequence of numbers shown by the **status** command; the gaps occur because dbx uses some of the numbers internally.

step

Executes up to the next line. The **step** command is different from the **next** command. The **step** command proceeds to the next line you can execute. As a result, if the current line contains a function call, execution stops on the first line of the called function. However, with the **next** command, execution continues until the next source line so it does not stop within the called function.

stop if *condition*
stop at *sourcelinenumber*[if *condition*]
stop in *function* [if *condition*]
stop *variable* [if *condition*]
Stops execution when one of the following conditions apply: the given condition is true, the given line is reached, the given subroutine or function is called, or the given variable is modified. The description of the **trace** command describes how to specify **stop** command arguments.

trace[in *function*][if *condition*]
trace *sourcelinenumber* [if *condition*]
trace *function*[in *function*][if *condition*]
trace *expression* at *sourcelinenumber* [if *condition*]
trace *variable* [in *function*] [if *condition*]
Displays tracing information when the program executes. A number is associated with each **trace** command. You must reference this number when using the **delete** command to turn off tracing.

If you specify the **in** clause with a function, tracing is in effect only within the given subroutine or function.

The variable condition is a Boolean expression that is evaluated prior to displaying the tracing information. If the result is false, the information is not displayed. Section 3.2.3 describes the operators used in conditional expressions.

The first argument—sourcelinenumber, function, expression, or variable—specifies what is to be traced as follows:

- If the argument is a source line number, the line is displayed immediately prior to execution. To specify source line numbers in a file other than the current file, precede the line numbers with the name of the file in quotes followed by a colon (:). For example, "xyz.c":17 specifies line number 17 in the file xyz.c.
- If the argument is a subroutine or function name, information is displayed every time the subroutine or function is called, telling what function called it, from what source line it was called, and what parameters were passed to it. In addition, a message noting the return is displayed and, if it is a function return, the value being returned is also displayed.
- If the argument is an expression with an **at** clause, the value of the expression is displayed each time the identified source line is reached.
- If the argument is a variable, the name and value of the variable is displayed if there is any change to the value or the name of the variable. The previous value of the variable, prior to the change, is also displayed. Execution is much slower using this form of tracing.
- If the argument is omitted, all source lines are displayed before they are executed. Execution is much slower during this form of tracing.

unalias *name*
Removes the **alias** associated with name.

unset *name*
Deletes the debugger variable associated with the name.

up ***[count]***
Moves the current scope, which is used for resolving names, up the stack by the number of levels specified in count. The default value of count is 1. See the **down** and **func** commands.

use ***directory-list***
Sets the list of directories to be searched when looking for source files. You can also do this by using the **–I** option on your **dbx** command line.

whatis ***name***
Displays the declaration of the given name. You can qualify the name using the function names described under the **print** command in this section.

where
Displays a list of the active functions.

whereis ***symbol***
Displays the full qualification of the specified symbol for each occurrence of the symbol in all functions in the program. The order in which the symbols are displayed is not meaningful.

which ***symbol***
Displays the full qualification of the given symbol within the currently active function in the stack.

/expression/
Searches forward in the current source file for the specified expression.

?expression?
Searches backward in the current source file for the specified expression.

3.4.2 Machine-Level Debugging Commands

Machine-level commands allow you to examine instructions that result from the expansion of a VAX C statement.

This section provides expanded descriptions of machine-level commands in alphabetical order.

address, address / [mode]
[address] / [count] [mode]
symbol / count [mode]
Displays the contents of memory, starting at the first address and continuing up to the second address or until count items are displayed (the default count is 1). The address is a memory location expressed as a hexadecimal, decimal, or octal number. Note that registers 0-15 are denoted by $rn (n is the number of the register) or, for registers 12 through 15, by $ap, $fp, $sp, or $pc, respectively.

Symbolic addresses are specified by preceding the name with an ampersand (&). Addresses can be expressions made up of other addresses and the plus (+), minus (–), and indirection (unary *) operators.

A symbolic name (symbol) that has been assigned the value of a memory location can also be specified, with enclosing parentheses, to achieve the same effect as address. For example, if the pc contains the address 0x100, then the following two commands would both print 10 instructions starting at 0x100:

```
(dbx)  0x100/10 i
(dbx)  ($pc)/10 i
```

If you replace the address with a period (.), the address following the one most recently displayed is used. The mode option specifies how memory is to be displayed; if mode is omitted, the previous mode specified is used. You must include the slash (/) even if you do not specify count or mode. The default mode is X. The following modes are supported:

i Displays the machine instruction

d Displays a short word (16 bits) in decimal

D Displays a long word (32 bits) in decimal

o Displays a short word in octal

O Displays a long word in octal

x Displays a short word in hexadecimal

X Displays a long word in hexadecimal

b Displays a byte in octal

c Displays a byte as a character

s Displays a string of characters terminated by a null byte

f Displays a single-precision real number, *float*

g Displays a double-precision real number, *double*

nexti

Executes up to the next machine instruction. The **nexti** command is different from the **stepi** command. The **stepi** command proceeds to the next instruction. As a result, if the current instruction contains a function call, execution stops on the first instruction of the called function. In contrast, if you enter the **nexti** command, execution does not stop within the called function.

stepi

Executes up to the next machine instruction. The **stepi** command is different from the **nexti** command. The **stepi** command proceeds to the next instruction you can execute. As a result, if the current machine instruction contains a function call, execution stops on the first instruction of the called function. In contrast, if you enter the **nexti** command, execution does not stop within the called function.

stopi at *address*
stopi if *condition*
stopi at *address* if *condition*

Stops execution when one of the following conditions apply: the given condition is true, the given address is reached, the given subroutine or function is called, or the given variable is modified at an address. See the **stop** command description for information on how to specify **stopi** command arguments.

tracei [*address*] [if *condition*]
tracei [*variable*] [at *address*] [if *condition*]

Turns on tracing using a machine instruction address. If you enter the **tracei** command without arguments, the execution of the entire program is traced.

The variable condition is a Boolean expression that is evaluated prior to displaying the tracing information. If the expression is false, the information is not displayed. Section 3.2.3 describes the operators used in conditional expressions.

3.5 Sample Debugging Session

This section contains a sample debugging session for a VAX C program. Example 3–1 contains a listing file for a program that requires debugging. The program was compiled and linked without diagnostic messages. The error occurs in the program's calculations, not in its syntax.

The program was designed to generate a table of squares for all values between 1 and 10. However, each value is added to itself rather than multiplied by itself. This is an obvious error. For illustrative purposes, this section deals with the problem as if it were not obvious.

NOTE

This section does not address machine-level debugging techniques.

Example 3–1: Sample VAX C Program

```
.MAIN
V1.0

  1            #include "stdio.h"
113            #define STARTNUM 1
114            #define ENDNUM 10
115
116
117            main ()
118            {
119    1           int    Count;
120    1           long   Square;
121    1
122    1           printf ("Table of squares \n\n");
123    1
124    1           for (Count = STARTNUM; Count <= ENDNUM ; Count++){
125    2               Square = (long)Count + (long)Count;
126    2
127    2               printf ("Number: %d Square: %ld\n", Count, Square);
128    2               }
129    1           }
130
```

When you debug a program, you are trying to find out what is happening at key points in your program. To do this, you must be able to stop execution and examine program locations. The point at which you stop execution is called a breakpoint. Breakpoints are set with the **stop** command.

Use the **print** or **dump** commands to examine the contents of a location. After encountering a breakpoint, use the **cont**, **next**, **step**, **run** or **rerun** commands to resume program execution.

Example 3–2 shows a dialog for a terminal debugging session. Red print in the example indicates your input. The numbers are keyed to notes that explain the procedure.

Example 3–2: Sample Debugging Session

```
%❶ vcc -g square.c
%❷ dbx a.out
dbx version 2.0 of 10/24/86 5:11.
type 'help' for help.
reading symbolic information...
[using memory image in core]
(dbx)❸ func main
(dbx)❹ list
8       {
9            int    Count;
10           long   Square;
11
12           printf ("Table of squares \n\n");
13
14           for (Count = STARTNUM; Count <= ENDNUM ; Count++){
15               Square = (long)Count + (long)Count;
16
17               printf ("Number: %d Square: %ld\n", Count, Square);
(dbx)❺ stop at 17
[1] stop at "square.c":17
(dbx)❻ run
❼ Table of squares
❽ [1] stopped in main at line 17 in file "square.c"
   17        printf ("Number: %d Square: %ld\n", Count, Square);
(dbx)❾ print SQUARE
"SQUARE" is not defined
(dbx)❿ print Square
2
(dbx)print Count
1
(dbx)cont
Number: 1       Square: 2
[1] stopped in main at line 17 in file "square.c"
   17        printf ("Number: %d Square: %ld\n", Count, Square);

(dbx)print Square
4
(dbx)print Count
2
(dbx)cont
Number: 2       Square: 4
[1] stopped in main at line 17 in file "square.c"
   17        printf ("Number: %d Square: %ld\n", Count, Square);
(dbx)print Square
6
(dbx)print Count
3
(dbx)⓫ delete *
(dbx)⓬ cont
```

(continued on next page)

Example 3–2 (Cont.): Sample Debugging Session

```
Number: 3        Square: 6
Number: 4        Square: 8
Number: 5        Square: 10
Number: 6        Square: 12
Number: 7        Square: 14
Number: 8        Square: 16
Number: 9        Square: 18
Number: 10       Square: 20
execution completed
(dbx)⓭ quit
%
```

Key to Example 3–2:

❶ Create an executable module with the default name a.out. This command line uses the **vcc** command program to invoke the VAX C compiler and the linker. The **–g** option generates the necessary dbx information.

❷ Invoke the dbx debugger, specifying the name of the executable module.

❸ Establish main as the current function.

❹ Display the function main on your terminal screen.

❺ Insert a breakpoint at line 17. Critical information is available at this point in the program.

❻ Initiate program execution.

❼ The program generates this message during execution.

❽ Execution halts when it reaches the breakpoint at line 17.

❾ An attempt is made to display the value of the variable SQUARE. This variable does not exist. The variable Square does exist.

❿ Display the contents of the variable Square.

⓫ Remove all breakpoints after isolating the cause of the error.

⓬ Continue program execution until processing is completed.

⓭ Exit from the dbx debugger.

Chapter 4

Program Structure

This chapter introduces the basic features of C to the programmer experienced in other languages. The text provides detailed examples and short tutorials, as well as numerous 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 also describes VAX C function definitions, function declarations, and the following components of C program structure:

- Function definitions
- Function declarations
- Function prototypes
- Function parameters and arguments
- Program identifiers
- Blocks
- Comments
- VAX C language keywords
- Functionality similar to that provided by lint

4.1 C Programming Language Background

The C language is a general-purpose programming language that is manageable due its small size, flexible due to 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 using the PDP–11. The designers of the language describe C 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 meet the needs of any particular application. C manipulates and stores data by considering the similarities found in modern machine architecture. However, C is not as complex as assembly language and is not machine dependent. A program is considered portable if

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

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

you can compile and run its source program using several different compilers on several different machines. C is a language that yields highly portable programs.

There is no ANSI or other industry-wide standard for the C programming language, but there is a consistency of functionality between implementations. This consistency is necessary for C to be portable across systems, which is one of the most desirable features of the language. Not only must C source programs be portable, but the language features themselves must produce the same effects on all systems when you compile and run programs.

C was developed in a UNIX system environment, and was used to rewrite most of that operating system, so many standard methods of operation in C are related to UNIX. For instance, UNIX systems access files by a numeric file descriptor, so C implementations should provide functions to access files by file descriptor. In a UNIX system environment, you can expect a concise command structure, an ability to redirect output from one program or command to the input of another program or command, an ability to create asynchronous and synchronous subprocesses, and an ability to manipulate the operating system features without many restrictions and system safeguards.

Some standard C constructs include preprocessor directives and macros. In a UNIX system environment, a preprocessor completes the tasks designated in the preprocessor directives located in the source code before the compiler takes any action.

4.2 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, based on the language design of Brian W. Kernighan and Dennis M. Ritchie. However, this version of VAX C for ULTRIX also incorporates some C language extensions based on the current draft of extensions proposed by the ANSI committee that is reviewing the C language. The added extensions are those most likely to be adopted. Digital does reserve the right to change the VAX C language, particularly these extensions, if the ANSI committee does not adopt the proposed extensions. Finally, VAX C also includes unique features that work directly and efficiently with the VMS operating system environment.

If you decide to retain both portability and utilize the ULTRIX environment, you can use special constructs of VAX C and options of the **vcc** command (such as the VAX C preprocessor predefined symbols and the command line option **–V standard=portable**. These constructs allow you to execute some segments of code only when running on ULTRIX systems, and to execute other segments of code when running on systems other than ULTRIX systems. See Chapter 9 for more information about the preprocessor directives. See Chapter 2 for information about the **vcc** command line.

4.3 Writing Your First Program

Writing your first program in an unfamiliar language can be frustrating, particularly if the program is complex and you introduce numerous errors. Start with simple programs. Computers are efficient at processing numbers, so the first program presented here adds two numbers and stores the total in a variable. Example 4–1 shows how to code such a program in the VAX C language.

Example 4–1: Simple Addition in VAX C

```
❶ /*  This program adds two numbers and places the sum in   *
   *  the variable total.                                   */

❷ main()                           /* The function name "main" */
      {                            /* Begins function body     */
❸        int total;                /* Variable of type "int"   */
                                   /* Blank lines are allowed  */
❹        total = 2 + 2;            /* Answer placed in "total" */
      }                            /* Ends the function body   */
```

Key to Example 4–1:

❶ The text between the character sequences (/*) and (*/) provide comments. You cannot place comments within comments (that is, they cannot be nested), but you can place comments anywhere that white space can appear. White space is an area within the source code where blank spaces or blank lines may separate code. In the 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; if a user-specified main function does not exist, the first function in the program stream at the time external references are resolved is the default main function. The main_program option is VAX C specific and is not portable. For more information about the syntax and using the main_program option, see Section 4.8.1.

VAX C functions have methods of exchanging information using parameters and arguments. In the function definition of main, there are no parameters, as designated by the empty parentheses. In Example 4–1, the function main cannot receive information by means of parameters.

To specify parameters in a function definition, list the parameter identifiers within the parentheses and separate them with a comma (,). You must declare the parameters before the beginning of the body of the function. If you call a function from within function main (you normally would not call the main function from another part of your program), the function name is followed by a list of arguments delimited by parentheses and separated by commas. The number of arguments must correspond with the number of parameters in the function declaration. In Example 4–1, there are no function calls.

The function performs its task as determined by the statements found in the body, and may or may not return a value to the calling expression. The body of the function main is delimited by braces ({ }). They are like 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 setup of the function, you can omit the **return** statement, and its return value will remain undefined. If a function does not return a value, you can declare the function to be of data type **void**. For more information about functions, see Section 4.10.

❸ The variable total is declared and defined within the function main. You must declare all variables before referencing them within the program. These 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 having the value of 0.

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. You can specify all VAX C storage classes and may place the reserved word describing the storage class either before or after the reserved word that describes the data type in the variable declaration.

Data types and storage classes are very important when determining the scope of a variable. For more information about data types, see Chapter 7. For more information about storage classes, see Chapter 8.

The reserved words used to identify data types (such as **int** and **double**), storage classes (such as **auto** and **globalvalue**), statements (such as **if** and **goto**), and operators (such as **sizeof**) are called keywords. Keywords are predefined identifiers that cannot be redeclared. You cannot use these words to identify variables and functions in your programs. Keywords must be expressed in lowercase letters. Section 4.13 lists the VAX C keywords.

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 the variable total in your program, the reference also must be lowercase. For example, if you try to reference the variable Total, an error occurs; the compiler does not recognize the variable name due to the initial capital letter.

❹ The sum of 2 + 2 is stored in the variable total. This is done using a valid VAX C statement. You can use any valid expression as a statement by ending it with a semicolon (;). The identifier total is a declared variable. The equal sign (=) and the plus sign (+) are valid VAX C operators. The numbers being added are valid constants. For more information about the various VAX C statements, see Chapter 5. For more information about the VAX C operators, see Chapter 6.

4.4 Producing Input/Output

The C language includes no facilities to administer input and output (I/O). However, all implementations must have methods that allow programs and users to communicate. The lack of communication in Example 4–1 is inconvenient; there is no way to know if the program assigns the correct value of 4 to the variable total. You can use an ULTRIX System Library function to output the value of the variable total to the terminal. For more information about the ULTRIX System Library functions, see the *ULTRIX Documentation Set.*

Before you can execute any of the example programs in this manual, you must define, in the correct order, the libraries that the linker must search to resolve references to library functions. For example, to compile and link a source program that employs functions from the math library, you must link against the ULTRIX System Library by specifying /lib/libm with the **vcc** command line option **-lm**. For information about linking, see Chapter 2 and the *ULTRIX Documentation Set*.

VAX C 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 macro source code in the .h definition files provided with ULTRIX systems. For example, you can display the standard I/O function, stdio.h, at your terminal with the following command:

```
% cat /usr/include/stdio.h RETURN
```

If this command causes an error, see your system manager. It is a good idea to type or print all the .h files to see the macros and definitions provided with ULTRIX systems.

For more information about macros, see Chapter 9.

Example 4–2 shows that by using the ULTRIX **printf** function, a VAX C program can print a message to the terminal.

Example 4–2: Output of Information

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

#include <stdio.h>

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

Key to Example 4–2:

❶ The **printf** function writes to the standard output device (the terminal screen). The first call to the 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 percent sign (%) is replaced by the value of total, the minus sign (–) left-justifies the output, and the letter d forces the value of the argument to be expressed as a decimal number. The period (.) prints immediately after the value of total.

The output from Example 4–2 is as follows:

```
Here is the answer: 4.
```

If you want to print the value of total on a separate line, add the newline character (\n) to the string. Example 4–3 shows how to output on two lines.

Example 4–3: Output Using the Newline Character

```
/*  This program adds two numbers, stores the sum in the       *
 *  variable total, and then prints the answer on two          *
 *  separate lines on the terminal screen.                     */

#include <stdio.h>

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

The output from Example 4–3 is as follows:

```
Here is the answer...
4.
```

After writing a program that produces output, it is necessary to compile, link, and execute the program using the **vcc** command 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 functions). Running a program executes the image.

In the ULTRIX environment, a file is distinguished by a file name and by a file extension. Choose a file name that is easy to identify. The file extension should reflect the functionality of the file. For example, the file extension .c is the required source file extension for the VAX C compiler.

After you create and appropriately name your program, invoke the VAX C compiler to compile and link it, then execute the result, as follows:

```
%  vcc  -o addition addition.c RETURN
%  addition RETURN
Here is the answer...
4.
$
```

You may have to define more libraries to the linker to use C functions in your program. For more information about creating source code and the compilation process, see Chapter 2.

4.5 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, the **while** statement, and the **for** statement. For information about statements that loop until meeting a condition, see Chapter 5.

4.5.1 The if Statement

You can use the **if** statement to force the program to execute one or more VAX C statements when a specified condition exists. Example 4–4 shows a program using the **if** statement.

Example 4–4: Conditional Execution Using the if Statement

```
/*  This program asks the user to guess a letter.  The      *
 *  program tells whether the answer is correct or          *
 *  incorrect.  The program is hard coded to accept 'a' or  *
 *  'A' as the correct letter.                              */

#include <stdio.h>

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!");
      }
}
```

Key to Example 4–4:

❶ The standard I/O **getchar** function retrieves a character from the standard input device (the terminal); the program pauses, waiting for the user to type a character and to press the RETURN key. The **getchar** function retrieves one character and ignores any others that are typed.

❷ If the letter that the user types is either a or A, a message stating that the choice is correct prints. If any other letter is typed, 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 conditionally, you must enclose the statements within braces ({ }). A statement or statements enclosed within braces is called a block or a compound statement. The concept of blocks is important to determine the scope of variables. See Section 4.14 for more information about blocks.

Sample output from Example 4–4 is as follows:

```
% example4 RETURN
Guess which letter I'm thinking of!
B RETURN
You're wrong.
You'll have to try again!
```

4.5.2 The switch Statement

The **if** statement is not the only method of specifying statements to be executed given a certain condition. A **switch** statement can perform the same task as the **if** statement in Example 4–4, but it is particularly useful when many conditions must be tested. Example 4–5 shows a program using the **switch** statement.

Example 4–5: Conditional Execution Using the switch Statement

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

❶ #include <ctype.h>           /* Include proper module      */

  #include <stdio.h>

  main()
  {
     char  ch;

     printf("Guess what letter I'm thinking of!\n");
     ch = getchar();
     if (isupper(ch));        /* If ch is uppercase         */
❷    ch = _tolower(ch);       /* Convert "ch": lowercase    */
     switch(ch)               /* Examine "ch"               */
        {                     /* Body of switch statement   */

           case 'a' :
              printf("You're right!");
              return;

           default   :        /* Any other answer           */
              printf("You're wrong.\n");
              printf("You'll have to try again!");
        }
  }
```

Key to Example 4–5:

❶ When using the macro **_tolower**, you must include the definition module ctype.h in the compilation process. The module ctype.h is located in the ULTRIX System Library 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 (#). Do not end preprocessor directives with a semicolon (;). The pound sign must appear in column 1, the leftmost margin of your source file.

The module ctype.h is not the only module that contains macros and definitions used by the functions. There are several ways to include definitions in the program stream. For more information about the VAX C definition modules, see Chapter 9 and the *ULTRIX Documentation Set.*

❷ The compiler replaces the references to the **isupper** and **_tolower** macros with lines of C source code. When the program is run, the value of the variable ch is translated to lowercase but only if it was originally an uppercase letter. To see the macro definitions of **isupper** and **_tolower**, print the file /usr/include/ctype.h from the system library.

Output from Example 4–5 is as follows:

```
% example5 RETURN
Guess which letter I'm thinking of!
A RETURN
You're right!
```

The **switch** statement executes one or more of a series of cases based on the value of the expression in parentheses. If the value of variable ch is a, then the statements following the label case `'a'` are executed. In Example 4–5, the **_tolower** macro in the **if** statement translates any uppercase answers to lowercase letters, so there is no need to test for the uppercase letter A in the **switch** statement. When a case label matches the value of expression ch, 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 bottom of the **switch** statement. The statements following the default label are executed if the value of the expression does not match any of the other case labels. For more information about the **switch** statement, see Chapter 5.

4.5.3 Loops

In the previous examples, the user can only guess once during the execution of the program. To guess another letter, it is necessary to execute the program again. If you want to execute a segment of code repeatedly until a condition is met, you can use a loop. Some loops execute a block of statements, known as the loop body, a specified number of times. Some loops test for a condition first and then execute the body of the loop if the condition is true. Some loops execute the loop body and then test for a condition, which guarantees at least one execution of the body. In VAX C, this last loop is called the **do** statement. Example 4–6 shows that you can use the **do** statement to alter the letter-guessing program.

Example 4–6: Looping Using the do Statement

```
/*  This program plays the same guessing game as the      *
 *  other examples except that the user must guess until  *
 *  the answer is correct.  This is accomplished using a  *
 *  do statement.                                         */

#include <stdio.h>

#include <ctype.h>

main()
{
   char  ch;

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

(continued on next page)

Example 4–6 (Cont.): Looping Using the do Statement

```
     do                         /* Do the following ...        */
        {                       /* Beginning of loop body      */
           ch = getchar();
           if (isupper(ch));
           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');
  }
```

Key to Example 4–6:

❶ 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 not-equal-to operator (!=) presents the condition to be tested. The **while** expression translates as follows: "while the variable ch is not equal to `'a'`."

Output from Example 4–6 is as follows:

```
% 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 allows you to specify the number of times to execute the loop body. In the previous examples, it can be used to limit the number of guesses that the user may attempt. You can use other looping techniques to limit the number of guesses, but you must be responsible for incrementing a counter (the **for** statement increments automatically). Example 4–7 shows the use of the **for** statement.

Example 4–7: Looping Using the for Statement

```
/*  This program plays the same guessing game as the     *
 *  previous examples except that the user is limited to  *
 *  three guesses.  This is accomplished using a for      *
 *  statement.                                            */

#include <stdio.h>

#include <ctype.h>

main()
{
   char    ch;
   int      i;               /* A counter 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();
         if (isupper(ch));
         ch = _tolower(ch);
         switch(ch)
            {
               case 'a' :
                  printf("You're right!");
                  return;
               case '\n':
❷                 --i;
                  break;

               default  :
                  printf("You're wrong.\n");
                  if (i != 3)
                     printf("You'll have to try again!\n");
            }                 /* End of switch statement  */

      }                  .    /* End of for loop body     */
   printf("Sorry, you ran out of guesses!");
}
```

Key to Example 4–7:

❶ 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 the loop counter i to value 1. The second expression establishes an upper bound; the value of variable i cannot 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, 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 are not counted as a guess.

Sample output from Example 4–7 is as follows:

```
% example7 RETURN
Guess which letter I'm thinking of!
You have three guesses.  Make them count!
B RETURN
You're wrong.
You'll have to try again!
```

```
C RETURN
You're wrong.
You'll have to try again!
U RETURN
You're wrong.
Sorry, you ran out of guesses!
```

4.6 Values, Addresses, and Pointers

In VAX C, every variable has two types of values: a memory location and a stored object. In VAX C, an lvalue is the variable's address in memory, and an rvalue is the stored object. Consider the following assignment statement:

```
put_it_here  =  take_this_object;
```

This assignment statement is not very different from statements in other programming languages, but consider 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 rvalue 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:

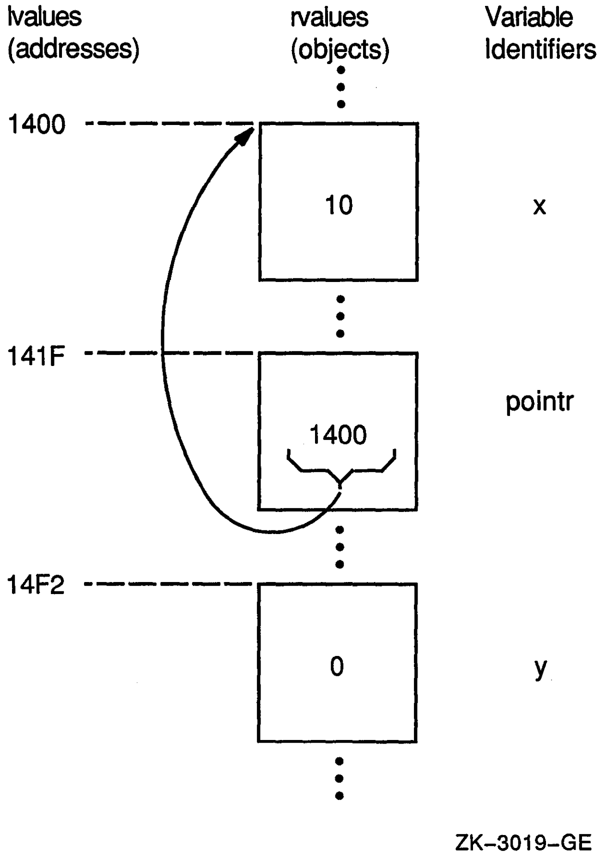
```
static  int *pointr;              /* Declarations            */
static  int x = 10, y = 0;

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

The rvalue of the variable pointr is the lvalue of variable x. 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 the operator, translates as follows: take the lvalue of this variable instead of its rvalue.

The **static** keyword specifies the **static** storage class. See Chapter 8 for information about **static** and other storage-class specifiers and modifiers.

Figure 4–1 shows the difference between rvalues and lvalues.

Figure 4–1: rvalues, lvalues, and Assigning Pointers



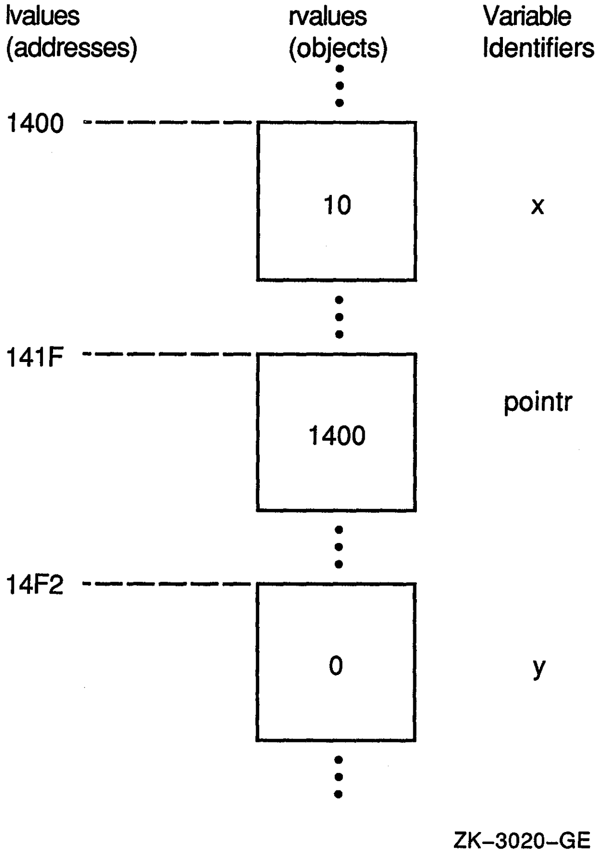
The value of the variable pointr contains the address of variable x. Remember that the location of variables in memory and the order in which the compiler processes them is unpredictable and left to the discretion of the compiler.

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

```
y = *pointr;
```

The asterisk (*) is the VAX C indirection operator; the value 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 4–2 shows the status of the variables after you execute the last code example.

Figure 4–2: The Indirection Operator in Assignments



For more information about pointers, see Chapter 7.

4.7 Aggregates

The variables used in the previous examples were either pointers or single objects that could be manipulated, in their entirety, in an arithmetic expression. These types of variables are called scalar variables. The VAX C data structures (arrays, structures, and unions) are called aggregates. Aggregates are comprised of segments called members. Members are sections of the structure that you can declare to be either a scalar or aggregate data type.

4.7.1 Arrays and Character Strings

An array is a data structure whose members are of the same type. Members of arrays can be any of the scalar or aggregate data types.

VAX C represents character strings 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 the string Hello to make the string six characters long. When assigning strings to character-string and array variables within the program, you must use the C string-handling functions **strncpy** or **strcpy**. The following example illstrates the use of character strings and arrays.

Example 4–8: Character String Constants and Arrays

```
/*  This program plays the same guessing games as the          *
 *  previous examples except that it uses character            *
 *  string constants and arrays.                               */

#include <stdio.h>

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 Example 4–8 is as follows:

```
% example8 RETURN
Guess which letter I'm thinking of!
B RETURN
You're wrong.
You'll have to try again!
```

For more information about arrays and character strings, see Chapter 7.

4.7.2 Structures and Unions

Structures and unions are aggregates whose members can be of different types. Structures and unions are declared using the **struct** and **union** keywords, 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 you can use to declare structure or union variables of the same type elsewhere in the program. Members of structures and unions may be referenced as follows:

```
main()
{
   struct  optional_tag                /*  Tag = optional_tag  */
      {
         char   letter_1;
         char   letter_2;
         int    number;
      } characters = {'a', 'b', 59}; /* Variable = characters */

   characters.letter_1 = characters.letter_2;
}
```

You may reference members by using the structure or union variable name followed by a period (.) or followed by the member name. As in Example 4–8, 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 0.

Unions are declared in the same way as structures, but all members in a union begin at offset 0. Unlike structures, unions cannot be initialized. The size of the union in memory is as large as its largest member. When the single storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered. Example 4–9 shows the nature of unions.

Example 4–9: Single Storage Allocation of Unions

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

#include <stdio.h>

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);
}
```

The output from Example 4–9 is as follows:

```
% example9 RETURN
Jackson
M
Mackson
```

The **strcpy** function copies the second string into the array variable. When assigning values to smaller union members, the compiler does not fill the remaining space in the union with NUL characters (\0); that is, whatever was in memory at the time remains. For more information about structures and unions, see Chapter 7.

Example 4–10 shows a structure definition and its usage.

Example 4–10: Structures

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

#include <stdio.h>

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         */
      };

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

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

   for (i = 1; i <= letter.num_guesses; i++)
      {
         ch = getchar();
         if (ch == letter.small_a || ch == letter.capital_a)
            {
               printf("%s", message1);
               return;
            }
         else
            if (ch == letter.newline_ch)
               --i;
            else
               {
                   printf("%s\n", message2);
                   if (i != 3)
                      printf("%s\n", message3);
                }

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

Key to Example 4–10:

❶ 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**.

❷ The struct variable letter is declared using the tag storage. At the same time, 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 Example 4–10 is as follows:

```
% 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!
```

See Chapter 2 and Chapter 3 for information about developing programs on ULTRIX systems.

4.8 Function Definitions

You must declare or define functions that you want to call or use in a VAX C program. You may or may not need to declare user-defined functions before you call them. This depends on what type of value the function returns, and the position of the function definition within the program. This section explains the rules for defining functions, but you may want to read the discussion of declarations and definitions in Chapter 7 before proceeding.

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 4.11 for a detailed discussion of parameters and arguments.

Example 4–11 shows an example of two function definitions.

Example 4–11: 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.h>          /* 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");
```

(continued on next page)

```
                             /* 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  */
```

Key to Example 4–11:

❶ Program execution begins with function main. The left brace ({) signifies the beginning of the function body; the right brace (}) signifies the end of the body. The function body is any set of valid VAX C statements or declarations. The body usually includes one or more **return** statements, as shown here. A **return** statement specifies an expression whose value is returned to the calling function. If you omit the expression, the returned value is undefined in the calling function. If you omit the **return** statement, the function terminates when the right brace is encountered and its return value is undefined. See Chapter 5 for more information about the **return** statement.

❷ The **lower** identifier begins a new function definition; lower has the single parameter c_up. Function main has no parameters, but 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 here in the program should specify only the names of parameters, not the names of other variables used in the function body. See Chapter 7 for more information about data types.

For more information about the VAX C operators used in Example 4–11, see Chapter 6.

4.8.1 The 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 linker. In Example 4–11, the main function physically precedes the **lower** function, but the two function definitions can appear in reverse order. The word main is not a language keyword, so you may use it 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 about scope and storage classes, see Chapter 8.

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 if the program does not contain function main, and if 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.

4.8.2 Parameter List Declarations

Example 4–11 shows 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 want 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 4.10 for more information about the effect of function prototypes.

The second way to declare parameter data types is as follows:

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

For instance, if you need to declare parameters of different data types, your function definition may appear as follows:

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

If you are using the function prototype format in a function definition, you must supply both an identifier and a data type specification for each parameter. If you do not, the action generates an error message.

In a function definition, you have the following two options when specifying an empty parameter list:

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

An example using the **void** keyword is as follows:

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

4.8.3 Function Return Data Types

By default, all VAX C functions return objects of data type **int**. In Example 4–11, 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 definitions and declarations. The following example shows the definition of a function returning a character:

```
char letter( int param1, char param2, int *param3  )
{
   .
   .
   .
   return param2;
```

If a function does not return a value, or if you do not call the function within an expression that requires a value, you can define the function to return a value of type **void**. Using 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 shows 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;
}
```

4.8.4 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, 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 about variable-length argument lists, see the description of the standard include file varargs.h in the *ULTRIX Documentation Set*.

When using ellipses for variable-length argument lists, you must have at least one argument preceding the ellipses. The following definition is allowed:

```
function_name( double lower, ... )
{
    .
    .
    .
```

The following definition is not allowed:

```
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, instead of using the ellipsis notation. The following example shows such a construct:

```
function_name( va_alist )
va_dcl
{
    .
    .
    .
```

NOTE

> If you use function prototypes, use ellipses (. . .) within parameter lists so that the compiler does not compare varargs declarations (va_alist and va_dcl) with prototype data declarations. See Section 4.10 for more information about function prototypes.

4.9 Function Declarations

As in Example 4–11, you can call a function without declaring it, if the function's return value is an integer. If the return value is anything else, you may have to declare the function. Example 4–12 shows a case where you need to declare a function.

Example 4–12: Declaring Functions

```
#include <stdio.h>

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;
{
    .
    .
    .
}
```

Key to Example 4–12:

❶ Since the location of the function definition is located after the main function in the source code, and since the **lower** has a return type of **char**, you have to declare the function before calling it.

If the function definition of **lower** is located before the main function in the source code, despite **lower**'s return data type, you do not need to declare the **lower** before you call the function.

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

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

If the function'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 4.10 for more information about function prototypes.

4.10 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 compile-time argument checking similar to that found in the lint utility. See Section 4.16 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 )
{ }
```

You can also define the function as follows:

```
char  function_name( lower, upper, func, y )
int   lower;
int   *upper;
char  (*func)();
double y;
{ }
```

This function's identifier list includes an integer, a pointer to an integer, a pointer to a function returning a character, and a double floating point value. The type specifications are identical to the ones used in a parameter list located before the function body. For more information about interpreting complex declarations, see Chapter 7.

In each compilation unit in your program, you should determine where to place the corresponding function prototype. The position of the prototype determines the prototype's scope; the scope of the function prototype is the same as the scope of any function declaration. VAX C checks all function definitions, declarations, and calls from the position of the prototype to the end of its scope. If you misplace the prototype so that a function definition, declaration, or call occurs outside the scope of the prototype, the results are undefined.

Corresponding function prototype declarations are identical to the header of a function definition that specifies data types in the identifier list. Since prototypes are actually function declarations, end the prototype code with a semicolon (;). The following example shows 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 *, 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 shows the specification of a variable-length argument list:

```
char  function_name( int lower, ... );
```

You cannot omit data type specifications in a function prototype, nor can you have a variable-length argument list that is not preceded by at least one argument. The following prototypes are not allowed 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( ... );
```

Using 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 is specified within a block enclosing the function call or at the outermost level of the source file. If a prototype is in scope, the automatic widening of **float** arguments to **double** is not performed. However, the automatic widening of **char** and **short int** arguments to **int** is performed. If the number of arguments in a function definition, declaration, or call does not match the prototype, the statement generates the appropriate message.

If the data type of an argument in a function call does not match the prototype, VAX C tries 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 4.11).

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. You place the include preprocessor directives at the top of any applicable compilation units.

See Chapter 9 for more information about preprocessor directives. See Chapter 8 for more information about compilation units and scope.

4.11 Using Parameters and Arguments

VAX C functions can exchange information by using parameters and arguments. (In this guide, the term parameter means the variable within parentheses named in a function definition; the term argument means an expression that is part of a function call.) In Example 4–11, 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 the parameters and arguments of VAX C functions:

- The number of arguments in a function call must be the same as the number of parameters in the function definition. This number may be zero.
- In VAX C, the maximum number of arguments (and corresponding parameters) is 253 for a single function. The maximum length of an argument list is 255 longwords.
- Arguments are separated by commas. However, the comma is not an operator in this context, and the compiler may evaluate the arguments in any order. Do not expect function calls or other complicated expressions in the argument list to be evaluated in any particular order.
- In VAX C, 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.
- The types of evaluated arguments must match the types of their corresponding parameters. When a function is called, unless a function prototype is in scope, VAX C does not compare the types of the arguments with those of the corresponding parameters so it does not generally convert the arguments to the types of the parameters. Instead, all 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 about the cast operator, see Chapter 6.

- If you declare variables in the parameter declaration section that do not exist in the parameter list, these variables are treated as if they are 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**.

4.11.1 Function and Array Identifiers as Arguments

You can use a function identifier without parentheses and arguments. In this case, the function identifier evaluates to the address of the function. This method of referencing is useful when passing a function identifier in an argument list. You can pass the address of one function to another as one of the arguments.

If you want 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 4–13 shows when you must declare user-defined functions and how to pass functions as arguments.

Example 4–13: 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     */
        .
        .
        .
  }
```

Key to Example 4–13:

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 about parentheses in expressions and the indirection operator, see Chapter 6.

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. To access the first element of the array, you need to dereference the pointer. For more information about pointers, addresses, and dereferencing, see Chapter 7.

4.11.2 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.

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)
```

Example 4–14 shows a program called echo.c, which displays the command-line arguments that were used to invoke it.

Example 4–14: Echo Program Using Command-Line Arguments

```
/*  This program echoes the command-line arguments.             */

#include <stdio.h>

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 Example 4–14 using the following command:

```
%  vcc  -o ECHO echo.c RETURN
```

Sample output from Example 4–14 is as follows:

```
% ECHO   Long  "Day's"  "Journey into Night" RETURN
program: /usr/oneill/plays/ECHO
argument 1: Long
argument 2: Day's
argument 3: Journey into Night
```

4.12 Identifiers

Identifiers can consist of letters, digits, dollar signs ($), and the underscore character (_). Do not create identifiers with a length of more than 255 characters. If you do, the compiler truncates the name and generates a warning message.

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 most implementation-specific macros and keywords, and uses two preceding underscores to identify implementation-specific constants.

Upper- and lowercase letters specify different variable identifiers; that is, the compiler interprets abc and ABC as different variable names.

Use the 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 other identifiers and keywords in lowercase.

4.13 Keywords

Keywords are predefined identifiers. They cannot be redeclared. They identify data types, storage classes, and certain statements in VAX C. Many conventional words in VAX C programs are not keywords and can be redeclared. The notable examples are the names of functions, including main and the functions found in system libraries.

Keywords must be expressed in lowercase letters.

Table 4–1 lists the VAX C keywords.

Table 4–1: 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 scaler type
void	Function return type
variant_struct[1]	Variant structure
variant_union[1]	Variant union
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[1]	Definition of a global variable
globalref[1]	Reference to a global variable
globalvalue[1]	Definition or declaration of a global value
readonly[1]	Allocated in read-only program section
noshare[1]	Assigned NSHR program section attribute
_align[1]	Aligned on a specified boundary
Type Modifiers	
const	Object cannot be modified
volatile	Object cannot be assigned to a register
Statements	
goto	Transfers control unconditionally
return	Terminates a function and optionally returns a value to the caller
continue	Causes next iteration of the containing loop

[1] VAX C specific and nonportable.

(continued on next page)

Table 4–1 (Cont.): VAX C Keywords

Keyword	Meaning
Statements	
break	Terminates its corresponding **switch** or loop
if	Executes the following statement conditionally
else	Provides an alternative for the **if** statement
for	Iterates the next statement (zero or more times) under control of three expressions
do	Iterates the next statement (one or more times) until a given condition is false
while	Iterates the next statement (zero or more times) while a given expression is true
switch	Executes one or more of the specified cases (multiway branch)
case	Begins one case for **switch**
default	Provides the default case for **switch**
Operator	
sizeof	Computes the size of an operand in bytes

The following identifiers are not true keywords, but the VAX C compiler defines substitutions so do not redefine them:

```
vax         VAX
vaxc        VAXC
vax11c      VAX11C
unix
ultrix
bsd4_2
CC$gfloat
```

See Chapter 9 for more information about these identifiers.

4.14 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. This definition of a block may conflict with its definition in other languages. In VAX C, the terms block and compound statement are identical.

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 4–15 presents nested blocks and the differences in the scope of declared variables.

Example 4–15: Scope of Variable Declarations in Nested Blocks

```
   /*  This program shows how variables with the same       *
    *  identifier can be of different data types if they    *
    *  are 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;
         }
   }
```

Key to Example 4–15:

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 about data types, see Chapter 7. For more information about scope and storage classes, see Chapter 8.

4.15 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 (*/). You cannot nest comments.

4.16 Source Code Checking Functionality

The lint utility 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, when combined, offer much of the functionality found in the lint utility at compile time:

Feature	Description
-V standard=portable	When you use the **vcc** command to compile your source code, add this option to the command. The compiler flags constructs that may not be supported by other implementations of the C language.
Function Prototypes	The use of function prototypes allows VAX C to check the number and the data types of all arguments passed to functions. See Section 4.10 for complete information.

Chapter 5

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

5.1 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.

5.1.1 The null Statement

Use null statements to provide null operations in situations where the grammar of the language requires a statement, but the program requires no work to be done.

The syntax of the null statement is as follows:

;

You may need to use the null statement with the **if**, **while**, **do**, and **for** statements in cases where the grammar requires a statement body but the program requires no functional operation. The most common use of this statement is in loop operations, where all the loop activity is performed by the test portion of the loop. For example, the following statement finds the first element of an array known to have a value of 0:

```
for(i=0; array[i] != 0; i++)
   ;
```

See Section 5.2 and 5.4 for more information about the statements mentioned here.

5.1.2 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 may not exist in storage. Attempts to access such variables may cause a run-time error. See Chapter 8 for more information about automatic variables.

5.1.3 The label Statement

Labels are identifiers used to flag a location in a program, and to be the target of a **goto** statement.

The syntax of a label is as follows:

```
identifier:
```

Any statement can be preceded by a label. The scope of 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 containing the label. Labels are used only as the targets of **goto** statements.

5.2 Expressions and Blocks as Statements

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

5.2.1 The expression Statement

You can use any valid expression as a statement by terminating it with a semicolon (;). The following example shows an expression used as a statement:

```
i++;
```

This statement increments the value of the variable i. Note that i++ is a valid VAX C expression that can appear in more complex VAX C statements. See Chapter 6 for more information about the valid VAX C expressions.

5.2.2 The compound Statement

A compound statement in VAX C is often 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 example shows 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 a label on the block itself. The compiler-generated code allocates storage for the **auto** or **register** variables each time the block is entered normally; the storage allocations do not occur if a **goto** statement refers to a label inside the block or if the block is the body of a **switch** statement. For more information about storage classes, see Chapter 8.

All function definitions are compound statements.

5.3 Conditional Statements

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

5.3.1 The if Statement

An **if** statement executes a statement depending on the evaluation of an expression, and may or may not be written with an **else** clause. The syntax of the **if** statement is as follows:

```
if ( expression )
    statement
else
    statement
```

An example of the **if** statement is as follows:

```
if (i < 1)
   funct(i);
else
   {
      i = x++;
      funct(i);
   }
```

If the evaluated expression within parentheses is true (in the example, if variable i is less than 1), then the statement following the evaluated expression executes; the statement following the **else** keyword does not execute. If the evaluated expression is false, then the statement following the **else** keyword executes.

All logical operators define a true result to be nonzero. Therefore, the expression in any conditional statement may be a logical expression with predictable results (true or false; nonzero or zero).

When **if** statements are nested within **else** clauses, each **else** clause matches the most recent **if** statement that does not have an **else** clause.

5.3.2 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 result of the evaluating expression must be of data type **int**. (For more information about the data types, refer to Chapter 7.) The statement is typically a compound statement, where one or more **case** labels prefix statements that execute if the expression matches the **case**. The syntax for a **case** label and expression is as follows:

```
case constant-expression :
     statement[,statement, . . . ]
```

The constant-expression must 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 the expression matches a **case** label, 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 that is encountered is executed. (See Example 5–1.) You can terminate the **switch** statement by a **return** or **goto** statement; if the **switch** is inside a loop, you can terminate it with a **continue** statement. For more information about interrupting statements, see Section 5.5.
3. If the expression's value does not match any **case** label but there is a **default** case, the **default** case is executed. It need not be the last case listed. If a **break** statement does not end the **default** case and it is not the last case, the next case encountered is executed.
4. If the expression's value does not match any **case** label and there is no **default**, the body of the **switch** statement is not executed.

In general, you must use the **break** statement to ensure that a **switch** statement executes as expected. Example 5–1 uses the **switch** statement to count blanks, tabs, and newlines entered from the terminal.

Example 5–1: Using the switch Statement to Count Blanks, Tabs, and Newlines

```
/*  This program counts blanks, tabs, and newlines in text *
 *  entered from the keyboard.                             */

#include <stdio.h>
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);
}
```

Key to Example 5–1:

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 automatically passes control to the **while** loop if none of the counters is incremented.

Example 5–1 responds to the following input:

```
% example RETURN
Every good boy. RETURN
The quick brown fox. RETURN
Line with 2 TAB TAB tabs. RETURN
^D
```

The output is as follows:

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

If you omit the **break** statements, the output is as follows:

```
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 the variable 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.

If variable declarations appear in the compound statement within a **switch** statement, any initializations of the **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 not executed.

5.4 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, which guarantees at least one execution of the body. In VAX C, this loop is the **do** statement.

5.4.1 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 (;) separate the expressions. Note that a semicolon does not follow the last expression. A **for** statement performs the following evaluations:

- Expression-1 is evaluated once before the first iteration of the loop. It usually specifies the initial values for variables.
- 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.
- Expression-3 is evaluated after each iteration. It usually specifies increments for the variables initialized by expression-1.
- Iterations of the **for** statement continue until expression-2 produces a false (0) value, or until some statement such as **break** or **goto** interrupts.

The **for** statement is identical to the following syntax:

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

The VAX C compiler optimizes certain **for** statements for simple loops. Consider the following example:

```
for(i=0; i<15; i++)
     printf("%d\n", i);
```

When the incrementation is as simple as in the previous example, the compiler generates less macro code so efficiency increases. When possible, use **for** statements instead of **while** statements when the increment is small.

You can omit any of the three expressions in a loop. If you omit expression-2, the test condition is true; that is, the **while** in the expansion becomes **while**(x), where x is not equal to 0. If you omit either expression-1 or expression-3 from the **for** statement, that expression is effectively dropped from the expansion.

The following syntax shows an infinite loop:

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

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

5.4.2 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++;
   }
```

The previous example tests the value of the variable x. If variable x is less than 10, 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.

5.4.3 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.

5.5 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.

5.5.1 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;

5.5.2 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 identical to the **goto** label statement for each of the looping statements in the following syntax examples:

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

In the preceding syntax examples, a **continue** statement passes control to label. The **continue** statement is intended only for loops, not for **switch** statements. A **continue** inside a **switch** statement that is inside a loop causes continued execution of the enclosing loop after exiting from the body of the **switch** statement.

5.5.3 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** value to be of type **void**. For more information about the **void** data type and function return values, see Chapter 4.

Chapter 6

Expressions and Operators

An expression is any series of symbols that VAX C uses to produce a value. The simplest expressions are constants and variable names, which contain no operators and 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 6.9.1 for more information about these rules.

This chapter discusses the following topics:

- The lvalues and rvalues
- Primary expressions and operators
- An overview of the 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

6.1 The lvalues and rvalues

A variable identifier is one of the primary VAX C expressions. (See Section 6.2 for more information about primary expressions.) This type of expression yields a single value, the contents 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. The expression on the right side evaluates to the variable's rvalue while the expression on the left side evaluates to the variable's lvalue when performing an assignment.

The following syntax defines the VAX C expressions that either have or produce lvalues:

```
lvalue ::=
      identifier
```

```
primary [ expression ]
lvalue . identifier
primary -> identifier
* expression
( lvalue )
```

These expressions represent the following identifiers and references:

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

All lvalue expressions represent a single location in a computer's memory. For a graphic explanation of lvalues and rvalues, see Chapter 4.

6.2 Primary Expressions and Operators

Simple expressions are called primary expressions, which 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 identifiers are 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.

6.2.1 Parenthetical Expressions

An expression within parentheses has the same type and value as the same expression without parentheses. As in declarations, you can delimit any expression using parentheses to change the grouping, or associative precedence, of the operators in a larger expression.

6.2.2 Function Calls

A function call is a primary expression followed by parentheses. The parentheses can contain a list of arguments (separated by commas) or it can be empty. An undeclared function is assumed to be a function returning **int**. If you declare an identifier as a function returning **int**, but use the identifier in a context other than a function call, it converts to the address of a 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, as in the following example:

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

You can also use the **atof** identifier in other contexts without the parentheses, as follows:

```
dispatch(atof);
```

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

Functions can also be called using a pointer to a function. Consider the following pointer declaration and assignment:

```
double (*pfd) ( );
    .
    .
    .
pfd = atof;
```

To call the function, you can specify the following form:

```
result = (*pfd) (c);
```

VAX C also accepts a pointer to a function, as shown in the following form:

```
result = pfd (c);
```

While the first call to the function is valid, the second call to the function is simpler and requires fewer keystrokes.

6.2.3 Array References ([ ])

Use bracket operators ([ ]) to see 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. Consider the following statement:

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

This statement assigns a value of 10 to the element sample_array[9][4][0].

Consider the following statement:

```
sample_array = 10;
```

The statement assigns a value of 10 to the element sample_array[0][0][0]. A reference to an array name with no bracket operator is often used to pass the array's address to a function, as in the following case:

```
funct(array);
```

You can also use bracket operators to perform general address arithmetic, using the following form:

```
addr[intexp]
```

In the previous example, 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 0, the result is the address of the same object. If intexp is negative, the result is the address of a previous object. The expressions *(addr + intexp) and addr[intexp] are equivalent because both expressions reference the same memory location.

6.2.4 Structure and Union References

You can reference a member of a structure or union with either the period (.) or the right arrow (–>) operator.

A primary expression followed by a period and 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 about structures and unions, see Chapter 7.

The form for a pointer to a structure and union uses the right-arrow operator (–>), which is specified with a hyphen (–) and a greater-than symbol (>). A primary expression followed by a right arrow and 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 right-arrow operator must name a declared member of that structure or union. The result is a reference to the named member.

The primary expression in bboth cases can be either a pointer or an integer. If it is a pointer, VAX C assumes that it points to a structure where the name on the right is a member. If it is an integer, VAX C assumes that it is the absolute address of the appropriate structure in machine storage units. If you specify something other than a pointer to a structure or union, VAX C signals the QUALNOTSTRUCT informational message. If you point to a different structure or union type, VAX C signals the NONSEQUITUR informational message.

6.3 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 6–1 lists the VAX C operators.

Table 6–1: 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	a after increment
++ [postfix]	a++	a before increment
- - [prefix]	- -a	a 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
=	a = b	a (with 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 6.2 for more information)

Table 6–2 presents the precedence by which the compiler evaluates operations. Operators with the highest precedence appear at the top of the table; operators with the lowest precedence appear at the bottom. Operators of equal precedence appear in the same row.

Table 6–2: Precedence of VAX C Operators

Category	**Operator**	**Associativity**
Primary	( ) [] –> .	Left to right
Unary	! ~ ++ – – (type) * & **sizeof**	Right to left
Binary (multiplication)	* / %	Left to right
Binary (addition)	+ –	Left to right
Binary (shift)	<< >>	Left to right
Binary (relational)	< <= > >=	Left to right
Binary (equality)	= = !=	Left to right
binary (bitand)	&	Left to right
Binary (bitxor)	^	Left to right
Binary (bitor)	|	Left to right
Binary (AND)	&&	Left to right
Binary (OR)	| |	Left to right
Conditional	?:	Right to left
Assignment	= += –= *= /= %= >>= <<= &= ^= | =	Right to left
Comma	,	Left to right

Consider the following expression:

```
A*B+C
```

The identifiers A and B are multiplied first because the multiplication operator (*) is of higher precedence than the addition operator (+). The associative rule applies to each row of operators. The following expression evaluates as A divided by B with the result then divided by C, because the division operator evaluates from left to right:

```
A/B/C
```

6.4 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 types (**sizeof**)

6.4.1 Negating Arithmetic and Logical Expressions

Consider the syntax of the following expression:

```
- expression
```

This defines 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 following expression is the logical (Boolean) negative of the expression:

```
!  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.

6.4.2 Incrementing and Decrementing Variables

The object that the lvalue refers to in the following expression is incremented before its value is used:

```
++lvalue
```

After evaluating this expression, the result is the incremented rvalue, not the corresponding lvalue. For this reason, expressions that use the increment and decrement operators in this manner cannot appear by themselves on the left side of an assignment expression that needs an lvalue.

The object to which the lvalue refers in the following expression increments after its value is used:

```
lvalue++
```

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. For example, incrementing a pointer to **int** increments the value of the pointer by 4. 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 make assumptions about which expressions the compiler evaluates first. To avoid ambiguity, increment the variable in a separate statement.

6.4.3 Computing Addresses and Dereferencing Pointers

Consider the following syntax:

& variable

The expression results in the lvalue (address) of variable. You may not apply the ampersand operator (&) 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 unless you use **–V standard=portable**; if you do, the compiler issues an appropriate message.

In the special context of argument lists, you may apply the ampersand operator to constants. This use of the ampersand operator passes constants to user-defined functions that expect arguments to be passed by reference. This use is not recommended for other applications. It is VAX C-specific and not portable.

Since function identifiers and unqualified array identifiers are lvalues, you cannot apply the ampersand operator to these identifiers. If you apply the ampersand operator to function identifiers or to unqualified array identifiers, VAX C considers this a redundant use of the ampersand operator and generates the appropriate error message when the **–V**"STANDARD=NOPORTABLE" option is used.

When an expression evaluates to an address, as in the following example, the address is used to indirectly access the object that the address refers to:

```
* pointer
```

An expression using the indirection operator (*) evaluates to the object pointed to by a pointer or by an address-valued expression.

6.4.4 Calculating a One's Complement

Consider the following syntax:

```
~ expression
```

The result is the one's complement of the evaluated expression; it converts each 1-bit into a 0-bit and each 0-bit into a 1-bit. The expression must be integral (an integer or character). The compiler performs necessary arithmetic conversions.

6.4.5 Forcing Conversions to a Specific Type Using the 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 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 as follows:

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

Consider the following form of the abstract-declarator:

```
abstract-declarator( )
```

To avoid confusion with the previous form, the abstract-declarator may not be empty in the following form:

```
(abstract-declarator)
```

Abstract declarators can include the brackets and parentheses that indicate arrays and function calls. However, cast operations cannot force the conversion of any expression to an array, function, structure, or union. The brackets and parentheses are used in such operations as the following, which casts identifier P1 to pointer to array of **int**:

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

This kind of cast operation does not change the contents of P1; it only causes the compiler to treat the value of P1 as a pointer to such an array. For example, casting pointers in this manner can change the scaling that occurs when you add an integer to a pointer.

6.4.6 Calculating Sizes of Variables and Data Types (sizeof)

Consider the following syntax:

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 6.4.5 for more information about the cast operator.

6.5 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 (>>)

The following sections describe these binary expressions and operators.

6.5.1 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 6.9.1.

You can increment an array pointer by adding an integral variable to the address of an array element. The compiler calculates the size of one array element, multiplies that by the integer to obtain the offset value, and then adds the offset value to the address of the designated element. For example:

```
int arr[10];
int *p = arr;
p = p + 1; /* Increments by 4 */
```

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

When you add or subtract two **enum** constants or variables, the type of the result is **int**.

If you subtract two addresses of objects of the same type, the result is 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.

6.5.2 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 0.

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 0, then the following is always true:

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

6.5.3 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**. Consider the following example:

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

The result is the value 1, if both relational expressions have the same truth value, and 0 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 0. A null pointer is considered equal to 0.

Although different symbols are used for assignment and equality, (=) and (==), respectively, VAX C allows either operator in some contexts, so you must be careful not to confuse them. Consider the following example:

```
if (x=1) statement-1;
else     statement-2;
```

In the previous example, statement-1 always executes, since the result of assignment x=1 delimited by parentheses is equivalent to the value of x, which is equal to 1 or true.

6.5.4 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 relational operators group from left to right. However, note that the following statement compares the variable c with 0 or 1 (the possible results of a<b); it does not mean "if b is between a and c...":

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

In order to check that b is between a and c, you should use the following code:

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

6.5.5 Bitwise Operators

You may only use bitwise operators with integral operands, with variables of types **char** and **int** (all sizes). The compiler performs the necessary arithmetic conversions. The result of the expression is the bitwise AND (&), OR (|), or EXCLUSIVE OR, which is represented by XOR (^), of the two operands. The compiler always evaluates all operands. Figure 6–1 shows the effects of Boolean algebra when using the bitwise operators.

Figure 6–1: 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–GE

In Boolean algebra, VAX C compares values bit by bit. If you use the bitwise AND, and you compare a bit value 1 and a bit value 0, the result is 0. When you use the bitwise AND, both compared bits must be 1, as shown in Figure 6–1, for the result to be 1. When you use the bitwise OR, either bit value can be 1 for the result to be 1. When you use the bitwise EXCLUSIVE OR, either value, but not both, can be 1 for the result to be 1. Figure 6–1 shows the use of all three bitwise operators on two common values.

6.5.6 Logical Operators

The logical operators are AND (&&) and OR (| |). These operators guarantee left-to-right evaluation. The result of the expression (of type **int**) is either 0 (false) or 1 (true). If the compiler can determine the result by examining only the left operand, it does not evaluate the right operand. Consider the following expression:

```
E1 && E2
```

The result is 1 if both its operands are nonzero, or 0 if one operand is 0. If expression E1 is 0, E2 is not evaluated. Similarly, in the following expression, the result is 1 if either operand is nonzero, and 0 otherwise:

```
E1 || E2
```

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.

6.5.7 Shift Operators

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

```
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 the following expression is the value of expression E1 shifted to the right by E2 bits:

```
E1 >> E2
```

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 is greater than 32.

6.6 The 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. Consider the following example:

```
E1  ?  E2  :  E3
```

If expression E1 is nonzero (true), then E2 is evaluated. If E1 is 0 (false), E3 is evaluated. Conditional expressions group from right to left. The compiler makes conversions in the following order:

- If possible, the arithmetic conversions are performed on expressions E2 and E3, so that they will result in the same type.
- Otherwise, if expressions E2 and E3 are address expressions indicating objects of the same type, the result has that type.

- 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.

6.7 Assignment Expressions and Operators

An assignment is an expression as well as an operation. The result of an assignment expression is the value of the target variable after the assignment. You can use assignments 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). The following two assignment expressions are identical:

```
E1 += E2;
```

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

The expression E1 is evaluated once and must result in an lvalue. The type of the assignment expression is the type of E1, and the result is the value of E1 after the operation is finished. You must delimit some expressions in parentheses if the expressions possibly contain other operators of a lower precedence. For example, the following expressions produce identical results:

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

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

However, the following example produces different results:

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

In the following simple assignment expression, the value of expression E2 replaces the previous object of E1:

```
E1 = E2
```

In the following example, the expression adds 100 to the contents of a_number[1]:

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

The result of the expression is the result of the addition, which 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 operand before the assignment. (See Section 6.9.1.)

You can use the assignment operator (=) to assign values to structure and union members. You can assign one structure value to another if 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 can be a pointer, and the right operand (which must be integral) is converted in the same manner as the right operand in the binary plus (+) and minus (−) operations.

You can assign an address to an integer, an integer to a pointer, and the address of an object of one type to a pointer of another type. These assignments are simple copy operations, with no conversions. This usage can 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 exceptions to the spellings of the compound assignment operators shown in Table 6–2. The exceptions are as follows:

- If you write the operators in the order shown in Table 6–2, you can separate the two characters with blank spaces. The following two examples produce the same results:

  ```
  E1 += E2;
  ```

  ```
  E1 + = E2;
  ```

- You can also write the operators with the characters in reverse order, as in the following example:

  ```
  E1 =+ E2;
  ```

The second form generates an informational message 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 cases such as the following, which multiplies the result of expression E1 by the value of p:

  ```
  E1 =*p;
  ```

- Even with usage that follows the guidelines, you can introduce ambiguities, as in the following example:

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

6.8 The Comma Expression and Operator

If you separate two expressions with 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. The following example assigns the value 1 to variable R and the value 2 to variable T:

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

The type and value of the result of a comma expression are the type and value of the 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. Consider the following example:

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

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

6.9 Data-Type Conversions

VAX C performs data-type conversions in the following situations:

- When two or more operands of different types appear in an expression (including an assignment)
- When arguments other than long integers, addresses, or double-precision, floating-point numbers are passed to a function

- When arguments that do not conform exactly to the parameters declared in a function prototype are passed to a function
- When the data type of an operand is deliberately converted by the cast operator (See Section 6.4.5 for more information on the cast operator)

6.9.1 Converting Operands

The following rules (referred to as the arithmetic conversion rules) govern the conversion of operands in arithmetic expressions. Although they do not specify explicit conversions at the machine-language level, the rules govern in the following order:

- 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**.
- If either operand is **double**, the other converts to **double**, and that is the type of the result, unless you specify the **–f** option with the **vcc** command.
- If either operand is **unsigned**, the other converts to **unsigned**, and that is the type of the result.
- Otherwise, both operands must be **int**, and that is the type of the result.

The arithmetic conversions are performed on all arithmetic operands. Some operators, such as the shift operators (>> and <<) require integers as operands. If one operand is of type **float** or **double**, you cannot meet this requirement.

In previous versions of VAX C under the VMS operating system, floating-point arithmetic was carried out in double precision. Since the proposed ANSI C standard no longer requires this conversion, VAX C performs arithmetic in single precision if you specify the **–f** option with the **vcc** command. Whenever an operand of type **float** appears in an expression and **–f** is specified, 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 you convert an operand of type **double** 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 0.

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. Consider the following example:

```
int i;
char c;

i = 0xFFFFFF41;
c = i;
```

The result is to assign hex 41 ('A') to variable c. The compiler converts shorter signed integers to longer ones by sign extension.

When 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 that the pointer points to. See Section 6.5.1 for more information about scaling pointers.

6.9.2 Converting Function Arguments

The data types of function arguments are assumed to match the types of the formal parameters unless a function prototype declaration is present. In the presence of a function prototype, all arguments in the function invocation are compared for assignment compatibility to all parameters declared in the function prototype declaration. If the type of the argument does not match the type of the parameter but is assignment compatible, VAX C converts the argument to the type of the parameter. (See Section 6.9.1.) 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.

If there is no function prototype for the function, 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 6.4.5 for more information on the cast operator. For more information about manipulating argument lists, see Chapter 4.

Chapter 7

Data Types and Declarations

The values of both constants and variables have data types. Data types specify the amount of storage required and how to interpret the data object in that storage space. This chapter discusses the following topics in respect to to data types:

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

7.1 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. You may define a symbol to represent the constant value. (For more information about symbolic representation of constants, see Chapter 9.) Constants, like all data in VAX C, have data types. The data type determines the amount of storage needed and determines how to interpret the stored object or constant value. The compiler determines the data type of constants by the way their values are represented in the source code.

7.2 Variables

You can also represent data in VAX C using variables, whose values can change throughout the execution of the program. You must declare all variables used in a program. When you declare a variable, you specify the data type of the stored object. In VAX C, an object is a value requiring storage.

Declarations determine the size of a storage allocation; definitions initiate the allocation of storage. You can declare and define variables. Most variable declarations are also definitions because storage is allocated at that point in the program. To declare a variable, specify the data type. To define a variable, assign the variable the proper storage class and place the variable declaration within the program structure. Also, if you can initialize a variable in the declaration, the variable is defined. For more information about variable definitions, scope, and storage allocation, see Chapter 8.

There are two kinds of variables: scalar and aggregate variables. Scalar variables have objects that you can manipulate 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 either a scalar or aggregate data type.

7.2.1 Data-Type Keywords

To declare or define variables, you need to know the VAX C keywords associated with each data type. Table 7–1 lists the VAX C data-type keywords by classification.

Table 7–1: VAX C Data-Type Keywords

Scalar Keywords	Aggregate Keywords	Other Type Keywords
int	**struct**	**void**
long	**union**	
unsigned	**variant_struct**	
short	**variant_union**	
char		
float		
double		
num		

In the following sections, the keywords and operators used to declare variables of given data types are listed in the section header for easy reference.

VAX C also supports the **const** and **volatile** type modifiers. For more information about these type modifiers, see Chapter 8.

7.2.2 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 on 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 may contain operators that declare a pointer, function, or an 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 a single expression or a comma-list of one or more expressions in braces.

Consider the following example:

```
int  var_number = 10;
```

The declaration both declares and defines the integer variable, var_number, which has an initial value of 10. The **int** keyword specifies the amount of storage needed on a VAX system for an integer. The identifier var_number follows. The equality operator (=) initializes the variable with the literal constant 10; for the initialization to take place, storage is allocated and the variable is defined. Declarations must end in a semicolon (;).

The variable declaration in the previous example was not difficult to interpret, but even experienced VAX C programmers have difficulty interpreting complex variable declarations. See Section 7.15 for more information about interpreting the VAX C variable declarations.

7.3 Integers (int, long, short, char, and unsigned)

You can declare integer variables with the **int**, **long**, **short**, **char**, and **unsigned** keywords. The following is an example of an integer declaration:

```
int x;
```

Character variables are declared with the **char** keyword. An example of a character declaration with the initialization of a character variable is as follows:

```
char ch = 'a';
```

Table 7–2 specifies the sizes and ranges of integers.

Table 7–2: The 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

(continued on next page)

Table 7–2 (Cont.): The Size and Range of VAX C Integers

Keyword	Size	Range
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.

7.3.1 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), and A to F (also for hexadecimal integers).

Integer constants can also include an optional suffix consisting of the characters x, X, l, L, u, or U. Characters x and X specify hexadecimal integers. Characters l and L specify **long** integers (of 4 bytes or 1 longword). Characters u and U specify **unsigned** integers. Characters l or L and u or U can be combined to specify an **unsigned long** integer.

On some other implementations of the C language, values of the **int** data type require 16 bits of storage. On VAX architecture, values of the **int** data type require 32 bits of storage, the same amount of storage as values of the **long** data type. VAX C supports the L suffix only for the sake of program portability.

You can specify integer constants 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 0, it is assumed to be octal; if it begins with 0x, it is assumed to be hexadecimal.

In octal constants, the digits 8 and 9 have the octal values 010 and 011, respectively. For instance, the octal number 039 is equal to $3 * 8 + 9$, or decimal value 33; the octal number 080 is equal to $8 * 8 + 0$, or, decimal value 64.

Even though VAX C supports the digits 8 and 9 in octal constants, avoid using these octal constants to be compatible with other implementations of the C language.

Integer constants must not include a decimal point; constants with a decimal point are of type **double**. Integer constants that exceed a longword are treated as programming errors.

Character constants such as 'a' and '$' are also valid integer constants. Their integer values in VAX C are the values of the corresponding ASCII codes.

Some examples of valid integer constants are as follows:

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

The following examples show invalid integer constants:

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

7.3.2 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 97. If the value of a character constant is not large enough to fill 32 bits of memory, the compiler stores the character or characters in the low-order byte(s) and pads the remaining bytes with NUL characters (`'\0'`).

Character constants do not have to be single characters, as shown in the following example (please note that this is VAX C specific, and not portable):

```
int l_word = 'a:cd' /*  This constant contains 4 characters    */

printf("%c\n", l_word);
printf("%.4s", &l_word); /*  String with maximum 4 characters  */
```

Sample output from the previous example is as follows:

```
%  example RET
a
a:cd
%
```

If you print variable l_word as a character, the **printf** function prints only the character located in the low-order byte of the integer allocation. To print all of the characters in the longword allocated to the variable, you have to print the variable as a string and pass the address of the integer variable as an argument. If you print the integer variable as a string, be sure to specify a precision of at most 4, since you can never be sure if the next byte in the string is a terminating 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 want 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 7.11 for more information about character-string constants.

7.3.3 Escape Sequences

In VAX C, escape sequences are character strings that represent a single printing or nonprinting character. The term escape sequence does not designate a string beginning with the ASCII character ESC, as in VT100 escape sequences. Table 7–3 presents the escape sequences that specify the nonprinting characters, the apostrophe, and the backslash (\ ).

Table 7–3: 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 specifies any byte value (usually an ASCII code), where the digits ddd are one to three octal digits. The octal digits are limited to 0 through 7. A common use is to specify the ASCII NUL character, as follows:

```
'\0'
```

Similarly, the form \xddd specifies any byte value (usually an ASCII code), where the digits ddd 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) in octal and hexadecimal digits, respectively.

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

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

7.4 Floating-Point Numbers (float and double)

When declaring floating-point variables, you determine the amount of precision needed for the stored object. In VAX C, you can have either single-precision or double-precision variables. If you choose single precision, you use the F-floating format. If you choose double precision, you have the choice of using either the D_floating or G_floating formats.

Table 7–4 specifies the sizes and ranges of real numbers.

Table 7–4: The Size and Range of C Floating-Point Numbers

Keyword	Size	Range	Precision
float	32 bits	$0.29 * 10^{-38}$ to $1.7 * 10^{38}$	7 decimal digits

(continued on next page)

Table 7–4 (Cont.): The Size and Range of C Floating-Point Numbers

Keyword	Size	Range	Precision
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

Use the **float** keyword to declare a single-precision, floating-point variable, represented internally in the VAX F_floating point binary format.

The **double** keyword declares a double-precision, floating-point variable. You can use the **double** and **long float** keywords interchangeably. However, do not use **long float** 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 **–Mg** option on the compile command. (For more information about the compilation command line, see Chapter 2.) Do not link modules compiled with the D_floating representation with modules compiled with the G_floating representation.

7.5 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.

Floating-point constants can also include an optional suffix consisting of the characters l, L, f, or F. Constants without suffixes, or with the l or L suffix, are of type **double**. Constants with the f or F suffix are of type **float**.

The following examples are floating-point constants:

```
3.0e10
3.0E-10
3.0e+10
3E10
3.0
.120e2
.120
```

7.6 Pointers (*)

Pointers in VAX C are variables that contain the 32-bit addresses of other objects. They are declared with the asterisk operator and the data type of the object that it points to, as in the following example:

```
int *px;
```

Identifier px is declared as a pointer to a variable of type **int**; the construct *px is treated as a variable of type **int**. An expression such as *px yields the integer that px points to.

Unless an **[extern]** or **static** pointer variable is initialized, it is a null pointer. A null pointer is a pointer variable that has been assigned the integer constant 0. The contents of an uninitialized **auto** pointer are undefined.

In certain arithmetic expressions, the compiler uses the size of the object of the pointer. For example, if px is a pointer to an integer, px + 1 evaluates to the next integer address, 4 bytes after px. If px is a pointer to **char**, px + 1 yields the next **char** address, 1 byte after px. The compiler uses the type of the pointed object to scale the arithmetic.

A different result occurs with an expression such as the following example:

```
*px + 1
```

This expression evaluates to the value of the object that px points to plus one.

Some contexts may require a pointer of a particular type. This is 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 you can use the asterisk in any sort of declarator, you can have pointers to scalars, to functions, to other pointers, to structures, and so forth.

Use the ampersand (&) operator to take the address of an object. Consider the following example:

```
px = &x;
```

This statement assigns the address of variable x to pointer px. After an assignment such as this, a reference to *px yields the value of x.

Do not apply the ampersand operator to constants, to **register** variables, to function identifiers, or to array identifiers. Though the compiler will allow it, it is not portable.

The compiler stores constant values in a read-only program section (psect), so that 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 routines. For more information about psects, see Chapter 8.

If you do apply the ampersand to **register** variables, the optimizing section of the compiler prevents 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.

7.6.1 void Pointers

The **void** pointer is a pointer that does not have a specified data type to describe the object to which it points. In effect, this is a generic pointer. (In the past, VAX C programmers have used **char** * to define generic pointers; this practice is now discouraged for portability reasons.)

You can assign a pointer of any type to a **void** pointer without a cast (see Section 6.4.5 for more information on the cast operation). For example, you can use this type of pointer in function calls, in function arguments, or in function prototypes when the parameter or return value is a pointer of an unkown type. Consider the following example:

```
main()
{
   void  *generic_pointer;
   .
   .
   .
/* If the return value can be a pointer to many types . . .   */
   generic_pointer = func_returning_pointer( arg1, arg2, arg3 );
   .
   .
   .
}
```

The following statements are also valid:

```
main()
{
float *float_pointer;
void  *void_pointer;
   .
   .
   .
float_pointer = void_pointer;
/*  Or . . .    */
void_pointer  = float_pointer;
   .
   .
   .
```

See Section 4.8.2 for information about using **void** in function definitions.

7.7 Enumerated Types

An enumerated type is a user-defined data type that is not derived from the fundamental types. Each listed enumerator is associated with an incremented integer constant starting with 0. The following example shows the declaration of a variable and an enumeration type, or tag:

```
enum  shades
   {
      out, verydim, dim, prettybright, bright
   }  light;
```

This declaration defines the variable light to be of an enumerated type shades. The variable can assume any of the enumerated values.

The tag shades becomes the enumeration tag of the new type; out, verydim . . . bright are the enumerators with values 0 through 4. These enumerators are the constant values of the type shades and can be used wherever constants are valid.

If you have declared the tag, 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 have unique spellings. VAX C allows forward references to **enum** tags that have not been declared yet in the source code, but are declared further 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. Consider the following example:

```
enum spectrum
   {
      red, yellow = 4, green, blue, indigo, violet
   }  color2;
```

This declaration gives red, yellow, green, blue, . . . , the values 0, 4, 5, 6, . . .

Examining the value of a variable like color2 displays an integer, not a string such as red or yellow. They are stored internally as integers, but you should regard enumerated data types as being distinct from the fundamental types.

Type mismatches between the enumerated and fundamental types, or between different enumerated types, are errors. The following example is not valid:

```
enum
   {
      red, orange, yellow, green, blue, indigo, violet
   }  color1;

enum  illum
   {
      out, verydim, dim, prettybright, bright
   }  light;

light = red;
```

The enumerators red and light have different enumerated types.

The following example is also invalid:

```
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;
```

The cast operation (enum illum) causes the compiler to treat **enum** constant red and integer constant 1 as values of enumerated type illum.

Variables and enumerators of enumerated types take on various storage classifications when used with the **globaldef** and **globalref** storage-class keywords. For more information about using these storage-class keywords with enumerated types, see Chapter 8.

7.8 Arrays ([ ])

You declare arrays with the square bracket operators ([ ]), as in the following declaration of a 10-element array of integers called table_one:

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

The **int** type specifier 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 can 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. It is not recommended that you use array subscripts as shown in the following example:

```
int table_one[10];

table_one[10]  = 69;
table_one[5]   = table_one[11];
```

VAX C supports multidimensional arrays, which are arrays declared as an array of arrays. Consider the following example:

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

Variable table_one is a 2-dimensional array containing 20 integers. You can use C operators to form expressions with specific elements of an array, as follows:

```
++table_one[0][0];         /* Increment first element           */
```

In VAX C, arrays are stored in row-major order. The element table_one[0][0] immediately precedes table_one[0][1], which in turn immediately precedes table_one[0][2].

When you declare an array, either single- or multidimensional, the integer constant is optional in the first pair of brackets. Omitting the constant expression is useful in the following cases:

- If the array is external, its storage is allocated by a remote definition. Therefore, you can omit the constant expression 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 about external data declarations, see Chapter 8.

- If the declaration of the array includes initializers, you can omit the size of the array, as 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 you use the array as a function parameter, it must be defined in the calling function. However, the declaration of the parameter in the called function can omit the constant expression within the brackets. 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 to use an array in this manner:

  ```
  main()
  {
                            /* Initialize array        */
     static char arg_str[] = "Thomas";
     int    sum;
     sum = adder(arg_str); /* Pass address of array     */
        .
        .
        .
  }

  /*  Function adds ASCII values of letters in array    */

  adder(param_string)
  char param_string[];
  {
      int i, sum = 0;       /* Incrementer and sum      */
                            /* Loop until NUL char      */
      for (i = 0; param_string[i] != '\0'; i++)
         sum += param_string[i];
      return sum;
  }
  ```

  After 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.

7.9 Initializing Arrays

When initializing array elements, separate the values with a comma and delimit the comma list with braces ({ }). The rules for specifying a comma-list are as follows:

- If the initializer for an array begins with a left brace ({), then the following comma-list provides initial values for the array elements. The list of initializers can end with a comma, which is ignored. The number of initializers cannot be greater than the number of elements.
- If the initializer does not begin with a left brace, then only enough elements are taken from the initializer list to supply values to the array's elements. In this case, there can be more initializers than there are elements, and any remaining values in the list are left to initialize the next aggregate.

Initialize a single-dimension array as follows:

```
static int ex_array[5] = { 1, 22, 333, 4444, 55555 };
```

Initialize a multidimensional array as follows:

```
static 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 way to initialize 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 can omit elements in an initialization as follows:

```
static 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 0. All of the elements in the second row, which are not specified in the initialization, are initialized with 0. For more information about the static storage class, see Chapter 8.

NOTE

> You cannot initialize array elements without initializing all preceding elements. The following initialization is not valid:
>
> ```
>             example[3] = { 1 , , 3 };
> ```
>
> You must initialize the first and second elements before initializing the third.

7.10 Character-String Variables (char * and char [ ])

VAX C treats character strings as arrays; they are treated as the address in memory of the first character in the string. There are several ways to declare character-string variables. You can declare a character string by designating a pointer to the first character of that string, as in the following example:

```
char  *ex_string  =  "thomasina";
```

This expression copies an address, not a string, to variable ex_string. The object to which ex_string points, a character-string constant, ends with the NUL character (\0).

You can declare character-string variables as you would declare an array. For example:

```
char string_one[] = "thomasina";
char string_2[10] = "thomasina";
```

See Section 7.9 for more information about declaring and initializing character-string variables.

To copy one string to another, you must use the **strcpy** or the **strncpy** functions, as follows:

```
main()
{
   #include <stdio.h>
   char ex_string[26];
                                  /* Copy string into array   */
   strcpy(ex_string, "Character-string constant");
   printf("%s\n", ex_string);
         .
         .
         .
}
```

7.11 Character-String Constants

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

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

It has the 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 7.3.2 for more information.

The following rules apply to the characters used in character-string constants:

- You can use all characters, including the escape sequences, in strings.
- You must precede a quotation mark within a string with 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.

- You can use character strings to initialize variables of storage class **auto** as well as variables of other storage classes.

7.12 Structures and Unions (struct and 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 valid operators 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.

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

- 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 that matches the alignment appropriate to its type. For example, a short integer is aligned on a 2-byte boundary and a long integer is aligned on a 4-byte boundary. Gaps can appear in a structure as the compiler tries to achieve this alignment.

  Structures also observe the following restriction: the length of a structure must be a multiple of the greatest alignment requirement of any of its members. Thus, a structure that contains characters, short integers, and longwords will be a multiple of four in length to match the multiple of four bytes for the longword.
- 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. The rules for alignment of union members is the same as for structure members.
- Structures can be initialized; unions cannot.

The VAX C compiler aligns structure members on natural type boundaries by default. You can turn off member alignment with the **#pragma nomember_alignment** directive.

7.12.1 Declaring a Structure or Union

To declare structures and unions, use the **struct** or **union** keywords. You can follow the **struct** or **union** keywords with 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.

You can place declarations after the list of member declarations, which name and reserve storage for structure or union objects.

Structure or union declarations can take one of five forms, as follows:

- 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];
   };
```

- 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 ;
```

- 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;
```

- The fourth form uses the tag to see 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. Consider the following example:

  ```
  struct   person   george,mary;
  ```

- The fifth form uses only the **struct** or **union** keyword and the tag to override other identical tags in scope, and to reserve the tag for a later definition within a new scope. A definition within a new scope overrides any previous tag definition appearing in an outer scope. This use of declaring tags is called vacuous structure tag declaration. The declaration does not require the size of the structure as determined by the structure member list. Using such declarations, you can eliminate ambiguity when forward referencing tag identifiers. The following example shows such a case:

  ```
  struct  ambiguous {...};

  {
     struct ambiguous;   /*  Vacuous structure tag declaration. */
                         /*  Ignore previous tag currently in scope. */

     struct  inner
     {
        struct ambiguous *pointer;  /* Declare a structure pointer by */
         .                          /* forward referencing.          */
         .
         .
     };

     struct ambiguous    /* Vacuous declaration refers to this  */
     {...};              /* structure, not to the first one declared. */
  }
  ```

  In the example, the pointer to the structure defined using the 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;
```

7.12.2 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 other recent compilers, a structure or union reference must be completely qualified; that is, you must prefix a member name in a reference either with a pointer qualifier (pointer-name ->) or with the name of the structure or union and the names of all intervening members. Consider the following structure declaration:

```
main()
{
   struct
      {
         struct { int a1,a2,a3; }  mema;
         struct { int a1,a2,a3; }  memb;
      }  *pointer, structure;
   pointer = &structure;

   structure.mema.a1 = 1;          /* Unambiguous               */
   pointer->memb.a1 = 2;

   structure.a1 = 3;               /* Ambiguous: which "a1"?    */
   pointer->a1 = 4;
}
```

Member a1 must be uniquely qualified as being a member of structure mema or structure memb. 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 that is not recommended, the compiler assumes that the program has an error and issues diagnostic messages. The following checks apply to using 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 that it is not a member of, 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.

7.12.3 Initializing Structures

In structure declarations, initializers follow the structure variables, not the members. Separate initializing values with commas; delimit them with braces ({ }). See Section 7.9 for more information about comma lists.

An example that initializes 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 an **auto** structure, the contents of the uninitialized members are undefined. For more information about storage classes, see Chapter 8.

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 7–1 shows these initialization rules applied to an array of structures.

Example 7–1: The Rules for Initializing 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);
         }
}
```

Key to Example 7–1:

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

❷ You must delimit a structure initialization with braces.

❸ You must delimit an array initialization with braces.

Example 7–1 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
```

7.12.4 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 reproduces its members to the enclosing aggregate.

You declare variant structures and unions using the **variant_struct** and **variant_union** keywords. The format of these declarations is the same as regular structures or unions with the following exceptions:

- You must nest variant aggregates 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.

Consider the following code example, which 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, you need to specify all of the intermediate aggregate identifiers, as follows:

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

If you try to access member d without specifying the intermediate identifiers, you are accessing the incorrect offset from the incorrect structure. Consider the following example:

```
enclosing_struct.d
```

If you specify this notation, 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 example shows the same code using variant aggregates:

```
/*  The numbers to the right of the code present the byte offset  *
 *  from enclosing_struct.                                         */
struct  TAG_1
{
   int   a;            /* 0-byte  enclosing_struct offset */
   char  *b;           /* 4-byte  enclosing_struct offset */
   variant_union
   {
      int  c;          /* 8-byte  enclosing_struct offset */
      variant_struct
      {
         int  d;       /* 8-byte   enclosing_struct offset */
         int  e;       /* 12-byte  enclosing_struct offset */
      }  nested_struct;
   }
nested_union;
} enclosing_struct;
```

The members of variant aggregates nested_union and nested_struct are propagated to the immediately enclosing aggregate (enclosing_struct). The variant aggregates cease to exist as individual aggregates.

Since variant aggregates nested_union and nested_struct do not exist as individual aggregates, you cannot use tags in their declarations nor can you use their identifiers (nested_union and nested_struct) in any reference to their members. However, you can reuse the identifier names in other declarations and definitions within your program.

If you need to access member d, use the following notation:

```
enclosing_struct.d
```

If you use the following notation, unpredictable results occur:

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

If you use regular structure or union declarations within a variant aggregate declaration, VAX C reproduces 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 is of type **struct**, the following offsets will 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 so they are not portable.

7.12.5 Bit Fields

A structure member can consist of a specified number of bits, called a field, which can be named or unnamed. Use a colon (:) to separate the member's declarator (if any) from a constant expression that gives the field width in bits. No field can 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 0 causes the next member (generally another field) to be aligned on a byte boundary.

Bit fields must be **unsigned** or **int** data types. 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 ampersand operator (&) cannot be applied to fields, so there cannot be pointers to fields.

Sequences of bit fields are packed as tightly as possible. In VAX C, fields are assigned from right to left.

For example, consider 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 } ;
```

Member a.i. is aligned on the third byte (at bit 16), because structure elements of the **short** data type are aligned on word boundaries by default. (The preprocessor directive **#pragma member_alignment** can alter this default behavior.) Fields a.fld1 and a.fld2 are packed as tightly as possible in the second longword. The unnamed, 0-length field preceding member a.fld3 causes that field to be aligned on the next byte boundary (bit 40, in the second longword).

7.13 The void Keyword

The **void** keyword is a special data-type specifier that you use in function definitions and declarations for the following purposes:

- To specify a function that does not return a value
- To specify a function prototype with no arguments
- To specify a generic pointer

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'; }
```

The following example shows a function prototype of a function that accepts the address of a pointer to any object as its first and second arguments:

```
void memcopy (void *dest, void *source, int length);
```

For more information about the **void** data type and function prototypes, refer to Chapter 4.

7.14 The typedef Keyword

Use the **typedef** keyword to define an abbreviated name, or synonym, for a lengthy type definition. In such a declaration, the identifiers name types instead of variables. For example:

```
typedef char CH, *CP, STRING[10], CF();
```

In the scope of this declaration, CH is a synonym for character, CP is a pointer to a character, STRING is a 10-element array of characters, and CF is a function returning a character. You can use each of the type definitions in that scope to declare variables, as follows:

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

7.15 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 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. 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. Consider the following example:

```
x  =  *px;
```

In the case of simple declarators, this symmetry makes it easy to determine the type of an expression or the meaning of a declarator. Expression *px results in the integer object that px points to.

Complicated declarators can be difficult to interpret without some additional guidelines. The important one to remember is that the symbols used in declarators are C operators, subject to the usual rules of precedence and grouping (associative nature). In order of precedence, the operators used in declarators are as follows:

- 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.

- The unary asterisk (*), for indirection or "pointer to . . . ", which groups from right to left.

Consider the following example:

```
int  *x[];
```

Even this brief declaration may be confusing. Does it declare an array of pointers to integers, or a pointer to an array of integers? Since the brackets are of higher precedence, it follows that:

- *x[ ] is an integer
- x[ ] is a pointer to an integer
- x is an array of pointers to integers

You can interpret most complicated declarators and expressions quickly by using 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 enumerates 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 the meanings of the operators:

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

Consider the following example:

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

The breakdown is as follows:

- *x( )[ ] is of type **char**
- x( )[ ] is (pointer to) **char**
- x( ) is (array of) (pointer to) **char**
- x is (function returning) (array of) (pointer to) **char**

In the third step, the bracket operator is removed first because primary-expression operators are of equal precedence and group from left to right. That is, "( )[ ]" means "function returning array of", not "array of function returning . . . ".

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

As a general rule, when breaking down a declaration in this manner, remove the operators with the lowest precedence first. Then, if the operators are of equal precedence and group from left to right, remove the rightmost operator first; if they group from right to left, remove the leftmost operator first.

The declaration shown is semantically invalid; VAX C allows functions returning addresses of arrays, but not functions returning arrays. Perhaps the programmer intended to specify a function returning the address of an array of pointers to characters. To make the declaration valid, start at the bottom of a breakdown and work back to a valid declaration as follows:

- x is (function returning) (pointer to) (array of) (pointer to) **char**
- x( ) is (pointer to) (array of) (pointer to) **char**
- *x( ) is (array of) (pointer to) **char**
- (*x( ))[ ] is (pointer to) **char**
- *(*x( ))[ ] is **char**
- **char** *(*x( ))[ ];

In the final declaration, the first asterisk (since it groups right to left) applies to **char**.

Use parentheses in declarations along with the function parameter-list operator (( )) to change the binding of operators. For example, the outer parentheses introduced in the fourth step of the previous example prevent the brackets from binding to the inner set of parentheses.

Consider the following example:

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

The breakdown is as follows:

- (* (*x( )) [ ]) ( ) is **char**
- * (*x( )) [ ] is (function returning) **char**
- (*x( )) [ ] is (pointer to) (function returning) **char**
- *x( ) is (array of) (pointer to) (function returning) **char**
- x( ) is (pointer to) (array of) (pointer to) (function returning) **char**
- The identifier x is a function returning a pointer to an array of pointers to functions returning characters

Spaces are used in this example to separate the declarator into its component parts. Since spaces, tabs, and newlines are ignored by the parser, use them in declarations for clarity.

Tables 7–5 and 7–6 provide examples of legal and illegal VAX C declarations.

Table 7–5: Legal C Declarations

Declaration	Meaning
int i;	An **int**
int *p;	Pointer to **int**

(continued on next page)

Table 7–5 (Cont.): Legal C Declarations

Declaration	Meaning
int a[ ];	Array of **int**
int f( );	Function returning **int**
int **pp;	Pointer to pointer to **int**
int (*pa)[ ];	Pointer to an array of **int**
int (*pf)( );	Pointer to a function returning **int**
int *ap[ ];	Array of pointer to **int**
int aa[ ][ ];	Array of an array of **int**
int *fp( );	Function returning pointer to **int**
int ***ppp;	Pointer to a pointer to a pointer to **int**
int (**ppa)[ ];	Pointer to a pointer to an array of **int**
int (**ppf)( );	Pointer to a pointer to a function returning **int**
int *(*pap)[ ];	Pointer to an array of pointer to **int**
int (*paa)[ ][ ];	Pointer to an array of an array of **int**
int *(*pfp)( );	Pointer to a function returning pointer to **int**
int **app[ ];	Array of pointer to pointer to **int**
int (*apa[ ])[ ];	Array of pointer to an array of **int**
int (*apf[ ])( );	Array of pointer to a function returning **int**
*aap[ ][ ];	Array of an array of pointer to **int**
int aaa[ ][ ][ ];	Array of an array of an array of **int**
int ***fpp( );	Function returning a pointer to pointer to **int**
int (*fpa( ))[ ];	Function returning a pointer to an array of **int**
int (*fpf( ))( );	Function returning a pointer to a function returning **int**

Table 7–6: Illegal Declarations

Declaration	Meaning
int af[ ]( );	Array of function returning **int**
int *fa( )[ ];	Function returning an array of **int**
int ff( )( );	Function returning a function returning **int**
int (*paf)[ ]( );	Pointer to an array of a function returning **int**
int (*pfa)( )[ ];	Pointer to a function returning an array of **int**
int (*pff)( )( );	Pointer to a function returning a function returning **int**
int aaf[ ][ ]( );	Array of an array of a function returning **int**
int *afp[ ]( );	Array of a function returning a pointer to **int**
int afa[ ]( )[ ];	Array of a function returning an array of **int**
int aff[ ]( )( );	Array of a function returning a function returning **int**
int *fap( )[ ];	Function returning an array of pointer to **int**
int faa( )[ ][ ];	Function returning an array of an array of **int**

(continued on next page)

Table 7–6 (Cont.): Illegal Declarations

Declaration	Meaning
int faf( )[ ]( );	Function returning an array of a function returning **int**
int *ffp( )( );	Function returning a function returning pointer to **int**
int *ffa( )( )[ ];	Function returning a function returning pointer to an array of **int**
int fff( )( )( );	Function returning a function returning a function returning **int**

Chapter 8

Storage Classes and Allocation

The VAX C language defines a number of storage-class keywords that specify the scope of an identifier, the location of storage, and the lifetime of the storage allocation. Storage-class modifiers are keywords that you can use with the storage-class and data-type keywords that restrict access to variables. The order of the storage-class keyword, the storage-class modifier, the data-type modifier, and the data-type keyword within the variable declaration does not matter. Each declaration, by virtue of its position in the program source code, has a default storage class, but you may override the default by specifying a storage-class specifier or a storage-class modifier.

This chapter describes the following material:

- The scope of an identifier
- The location of storage
- The lifetime of storage allocation
- The internal storage class
- The static storage class
- The external storage class
- The global storage class
- The data-type modifiers
- The storage-class modifiers

8.1 Scope

The scope of an identifier is that portion of the program in which the identifier has meaning. An identifier has meaning if it is recognized by the compiler, or by the linker. The following sections explain the rules to follow so that your program identifiers will have meaning, to both the compiler and the linker, in all desired portions of your program.

All tags are subject to the same scope rules as other identifiers. A member of a structure or union may have the same name as a member of another structure or union; the scope of the member names can exist concurrently. However, when referencing one of the members in a section of the program where the scopes of both members are concurrent, specify to which structure or union the member belongs. For more information, see Chapter 7.

8.1.1 The Compilation and Linking Process

To understand scope, you must know the VAX C/ULTRIX 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 VAX C/ULTRIX equivalent is the compilation unit, not necessarily a single source file. The single, resultant object file has a file extension of .o by default.

The linker accepts the object file as input and clears up all external references, such as references to independently compiled external functions. Internally, segments of object code, such as the object file, are known to the linker as object modules. The object module has the same name (without an extension) as the object file by default.

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 about compiling and linking, refer to Chapter 2.

8.1.2 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, but a definition initiates the allocation of storage. Since declarations are often definitions, this section refers to both definitions and declarations as declarations. You may want to review Chapter 7 before reading the rest of this section.

The location of a declaration establishes the scope of an identifier. If a declaration is located inside a block, which is a segment of 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, and 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 internal storage-class identifiers; 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 **static** declaration is outside 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 external storage-class or global storage-class identifiers. If the declaration is outside 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 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 span object modules.

Table 8–1 lists the storage classes followed by the storage-class specifiers that you use to establish scope.

Table 8–1: VAX C Storage Classes and Storage-Class Specifiers

Storage Class	Specifiers
Internal	**auto**, **register**, absence of a specifier inside a block or function[1]
Static	**static**
External	**extern**, absence of a specifier outside of all functions
Global	**globaldef**, **globalref**, **globalvalue**

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

see the following sections for more information on storage classes:

- Section 8.3 discusses the internal storage class
- Section 8.4 discusses the static storage class
- Section 8.5 discusses the external storage class
- Section 8.6 discusses the global storage class

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 8.7 for more information about the data-type modifiers. See Section 8.8 for more information about the storage-class modifiers.

8.1.3 Lexical Scope and Link-Time Scope

When you use the storage-class specifiers and modifiers, be careful when positioning the definitions and declarations of your identifiers and keep the following two goals in mind to avoid error messages:

- Compile the program so that the compiler recognizes all identifiers in the compilation unit.
- Link the program so that the linker resolves all references to external data definitions.

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. In this guide, the term scope implies lexical scope.

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 8–2 lists the VAX C storage-class specifiers and shows both the link-time scope and the lexical scope implied by each specifier when used inside and outside of functions.

Table 8–2: 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

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

In Table 8–2, the **[auto]** notation specifies identifiers of the automatic storage class. If you do not include a storage-class specifier on a definition inside of a function definition, the object is **auto** by default. This notation is used throughout this manual to represent the automatic storage class, regardless of the presence of the **[auto]** specifier in the definition. In Table 8–2, the **[extern]** notation signifies identifiers of the external storage class. A single definition exists without using a storage-class specifier; other declarations, which use the **extern** specifier, may exist which reference that definition. This notation is used throughout this chapter. See Section 8.5 for more information about the external storage class.

8.1.4 Program Example

Determining the scope of **static**, external, and global symbols can be very difficult. In Example 8–1, consider the scope of the identifiers.

The following list specifies the variable identifiers in Example 8–1, and in which functions they can be accessed without compile-time errors:

Example 8–1: Scope and Externally Defined Variables

```
Compilation Unit 1                  Compilation Unit 2
------------------                  ------------------

globaldef int  GLOBAL_1;
          int  EXT_2;              int  EXT_1;
static    int  STAT;

f1()                               f3()
{                                  {
   globaldef int  GLOBAL_2;           extern int  EXT_2;
      .                                  .
      .                                  .
      .                                  .
}                                  }

extern int  EXT_1;                 globalref int  GLOBAL_2;

f2()                               f4()
{                                  {
   .                               globalref int  GLOBAL_1;
   .                                  .
   .                                  .
}                                     .
                                   }

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

Identifier	Scope
GLOBAL_1	This variable is defined outside all the functions in Compilation Unit 1, so you can access GLOBAL_1 in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit). In Compilation Unit 2, the declaration of this variable is located inside 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 function f1. In Compilation Unit 1, the scope of GLOBAL_2 only extends from the declaration of GLOBAL_2 to the end of function f1. In Compilation Unit 2, the declaration of this variable is outside all functions but is located after function f3. You can access the variable in functions f4 and f5 (from the point of the declaration to the end of the compilation unit).
EXT_1	This variable is declared outside all the functions. This declaration is a reference to the definition of the same variable in the other compilation unit. In Compilation Unit 1, you can access EXT_1 in function f2 (from the point of the declaration to the end of the compilation unit). In Compilation Unit 2, the definition of this variable is outside all the 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 all the functions. In Compilation Unit 1, you can access EXT_2 in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit). In Compilation Unit 2, the declaration of this variable is located inside the function f3. You can access EXT_2 from the location of this declaration to the end of function f3.
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 1, the variable is defined outside all the functions. You can access STAT, in Compilation Unit 1, in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit). In Compilation Unit 2, the separate variable is defined inside function f5; you can access STAT from this declaration to the end of function f5.

Another way to view the determination of scope is to consider the placement of the declaration as a matter of privacy. In Compilation Unit 2 in the previous example, identifier EXT_2 is made private to function f3 by placing the declaration inside the function body. If you want to keep a variable private to Compilation Unit 1, you can use a declaration using the **static** storage-class specifier. There is no way to access a variable declared with **static** in another compilation unit.

Using the **auto** and **register** internal storage-class specifiers or positioning a declarative with no storage class specifier inside a function declaration, assures privacy to the function. Internal storage is deallocated after execution of the function body. There is no way to access a variable declared with internal storage class in another function or compilation unit.

8.2 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, variables of the **[auto]** and **register** storage class 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.

Use program sections (psects), for permanent variables; the identifier's lifetimes extend through the course of the entire program. A psect represents an area of virtual memory that has a name, a size, and a series of attributes that describe the intended or permitted usage of that portion of memory. For example, the compiler places variables of the static, external, and global storage classes in psects. You have some control in determining the psects that contain 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 8–3 shows the location and lifetime of a variable when you use each of the storage-class keywords:

Table 8–3: Location, Lifetime, and the Storage-Class Keywords

Storage Class	Location	Lifetime
[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 psects that are created by your data declarations and VAX C.

8.3 Internal Storage Class

You can assign the internal storage class to identifiers using the **auto** and **register** storage-class specifiers or by placing a declaration that contains no storage-class specifiers inside a function body. The following sections describe these specifiers.

8.3.1 The auto Specifier

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. Within a function, **auto** is the default storage class; it is not necessary to explicitly specify it. That is, any variable (other than a function name) declared within a function without a storage-class specifier is given the **auto** storage class. Functions are of the external storage class by default. The code generated by the compiler contains instructions to allocate and deallocate **auto** storage by using machine registers and the run-time stack. Since new storage allocation occurs when you enter 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. This initialization cannot occur if control passes into a block by some other means, such as a **goto** statement or if the block is the body of a **switch** statement. For more information about the **switch** and **goto** statements, refer to Chapter 5.

NOTE

> The compiler can assign **auto** variables to machine registers, if possible. Otherwise, they are placed on the run-time stack.

Example 8–2 shows the reinitialization of two **auto** variables with the same name:

Example 8–2: Reinitializing auto Variables

```
   /*  This example prints the values of two distinct auto    *
    *  variables that have the same identifier.               */

   main()
   {
❶     int i, x = 2;
      printf("main: %d\n",x);

      for (i = 0; i < 1; i++)
         {
❷           int x = 3;
            printf("for loop: %d\n",x);
          }

      printf("main: %d\n", x);
   }
```

Key to Example 8–2:

❶ This definition of variable x extends through the entire function.

❷ This definition of variable x is limited to the **for** statement and supersedes the value of variable x in the surrounding function.

Output from Example 8–2 is as follows:

```
% example RETURN
main: 2
for loop: 3
main: 2
```

In Example 8–2, the variable x is defined twice within the main function, but the two variables do not conflict. While the **for** loop is executing, the variable x declared inside the block supersedes the variable x declared outside the block.

8.3.2 The register Specifier

Variables declared with the **register** storage class are similar to **auto** variables. You can only use the **register** internal storage class inside functions and blocks.

NOTE

> The **register** storage-class specifier is the *only* specifier that you can use in a parameter declaration.

A **register** variable differs from a variable of storage class **auto** in the way that compiler-generated program code allocates storage. The **register** storage-class keyword suggests that the compiler flag the variable for placement in a machine register. This does not guarantee that the program code will place the variable in a register. The compiler checks the following conditions to determine whether or not a variable is flagged to be placed in a register:

- If the variable is not used, the optimizer may remove it entirely.
- If the program is compiled with the **–V nooptimization** option, no variables are assigned to registers. The optimization phase of the compiler determines whether a variable is a valid candidate for a register. (Optimization is enabled by default.)
- If the program contains too many register candidates, not all of them are assigned to registers.
- If the compiler detects any use of the variable that may make it inappropriate for assignment to a register, the variable is not flagged. For example, if the compiler detects the application of the address-of operator (&) to a variable that is declared with the **register** specifier, the variable is not placed in a register.

8.4 The Static Storage Class

The static storage class allows you to create permanent storage for a variable using the **static** storage-class specifier in the variable declaration. If declared inside a function, its scope begins at the declaration and spans the body of the function. If declared outside of functions, its scope is limited to the compilation unit; you cannot access a variable of the static storage class from another compilation unit.

If no initialization is present in the declaration of a variable of the static storage class, the linker initializes the variable to 0. 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.

You can also define a function with the **static** storage class. A **static** function is not known to the linker and can be referenced only from within its defining module.

8.5 The 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 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** specifier, usually located in another compilation unit, is a reference to the original definition. This declaration extends the link-time scope of the variable into the second object module. If this declaration is inside a function, it extends the link-time scope from the point of the declaration to the end of the function. If this declaration is outside of a function, it extends the link-time scope from the point of the declaration to the end of the object module.
- You need not use external variable declarations (with the **extern** specifier) to see the definition of an external variable. Also, when necessary, you can use more than one **extern** declaration to reference the external definition.

Use the following rules to decide whether or not to use the **extern** specifier:

- If the variable is defined before it is referenced and the definition is in the same compilation unit, you do not need to declare the variable with the **extern** specifier.
- If the variable is defined after it is referenced, you need to first declare it with the **extern** specifier.
- If the variable is defined in a separate compilation unit, you must declare it with the **extern** specifier.

Consider the following example:

```
double D = 2.37;

main()
{
   extern int A;

   printf("a:\t%d\n", A);
   printf("d:\t%g\n", D);
}

int A = 5;
```

The main function in this program references two external variables, A and D. Since the variable D is defined before it is referenced, you do not need to declare it 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 see an external definition elsewhere in the program. External variables occupy storage in psects of the same name as the variable identifier.

Whenever the compiler encounters the first declaration of an identifier of the external storage class in a VAX C program, it creates and initializes the psect. In VAX C, you can use the **extern** specifier in a declaration that does not see 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.

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 psect with the NOWRT attribute. The **noshare** and **readonly** attributes are VAX C specific and are not portable.

8.6 The Global Storage Class

You can assign the global storage class to identifiers using the **globaldef**, **globalref**, or **globalvalue** storage-class specifiers. The following sections describe these specifiers.

8.6.1 The globaldef and globalref Specifiers

Use the **globaldef** specifier to define a global variable; use the **globalref** specifier to refer 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. Use **globalref** to see 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 its 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. You can also 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.

You may initialize variables declared with **globaldef**. Variables declared with **globalref** may not, since these declarations see 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 using the **readonly** or **const** modifier; unless the global variable is initialized when the variable is defined, its permanent value is 0.

Example 8–3 shows the use of global variables.

Example 8–3: Using Global Variables

```
  /*  This example shows how global variables are used      *
   *  in VAX C programs.                                    */

❶ int ex_counter = 0;
❷ globaldef double velocity = 3.0e10;
❸ globaldef {"distance"} long miles = 100;

  main()
  {
     printf("   *** FIRST COMP UNIT ***\n");
     printf("counter:\t%d\n", ex_counter);
     printf("velocity:\t%g\n", velocity);
     printf("miles:\t\t%d\n\n", miles);
     fn();
     printf("   *** FIRST COMP UNIT ***\n");
     printf("counter:\t%d\n", ex_counter);
❹    printf("velocity:\t%g\n", velocity);
     printf("miles:\t\t%d\n\n", miles);
  }

  /*  ----------------------------------------------------  *
   *  The following code is contained in a separate         *
   *  compilation unit.                                     *
   *  ----------------------------------------------------  */

  static ex_counter;
❺ globalref double velocity;
  globalref long miles;

  fn()
  {
     ++ex_counter;
     printf("   *** SECOND COMP UNIT ***\n");
     if ( miles > 50 )
        velocity = miles * 3.1 / 200 ;
     printf("counter:\t%d\n", ex_counter);
     printf("velocity:\t%g\n", velocity);
     printf("miles:\t\t%d\n", miles);
  }
```

Key to Example 8–3:

❶ The integer variable ex_counter is a variable of storage class **extern** in the first compilation unit. In the second compilation unit, a variable ex_counter is of storage class **static**. Even though they have the same identifier, the two ex_counter variables are different variables represented by two separate memory locations. The link-time scope of the second ex_counter is the module created from the second compilation unit. When control returns to the main function, the external variable ex_counter retains its original value.

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 it is stored in the user-specified psect distance.

❹ When the variable velocity prints after the function fn executes, the value will change. 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 Example 8–3 is as follows:

```
% example 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
```

8.6.2 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, and with other high-level languages that support global symbol definition.

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:

- Global variables correspond to global symbols declared in assembly-language programs but external variables correspond to FORTRAN common blocks.
- If you have a limited amount of storage available, you may psee use the **globalvalue** specifier (see Section 8.6.3) since it does not occupy storage in your program if you can express it in 32 or fewer bits; the external variables create program sections.
- You can declare a global variable, using **globaldef**, inside a function or block and, by using a **globalref** specifier, access the identifier from another compilation unit. With external variables, you must define the variable outside all functions and blocks, and then access that variable in other compilation units by using **extern** declarations.
- 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 that you can place the external variables (by default) and the global variables (optionally) in psects with a user-defined name, and to some degree, user-defined attributes. The compiler places external variables in psects of the same name as the variable identifier. 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.

The compiler places a variable declared using only the **globaldef** specifier and a data-type keyword into the $DATA psect.

8.6.3 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 see 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 see 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 see 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.

Example 8–4 shows the use of the **globalvalue** storage-class specifier.

Example 8–4: Using the globalvalue Specifier

```
/*  This program shows 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.h>                 /* Include proper module    */
globalvalue  FAILURE,  EOF;        /* Declare global symbols   */
```

(continued on next page)

Example 8–4 (Cont.): Using the globalvalue Specifier

```
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 Example 8–4, FAILURE and EOF are defined in the first module: the values are placed in the program stream. In the second module, FAILURE and EOF are declared so that their values can be accessed.

8.6.4 Global Enumerated Types

When you use the **globaldef** storage-class keyword with an **enum** definition, the enumerated constants in the definition are of the **globalvalue** storage class, and initialized as required by the program to form a properly ordered list of the values. Enumerated type variables are of the **globaldef** storage class.

When you use **globalref** with the **enum** keyword, all enumerated variables are of the storage class **globalref**, and the enumerated constants see global values of the same names as shown in the following example.

The first compilation unit includes the following sequence:

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

main()

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

The second compilation unit includes the following sequence:

```
globalref enum  light { dim, medium, bright }  light_val;

fnlv()
{
    if (light_val < bright ) printf("TOO DIM\n");
}
```

In the first compilation unit, the **enum** definition establishes light_val as a **globaldef** of the enumerated type light. It also establishes the ordered list of enumerated global values dim, medium, and bright.

The **globalref** declaration in the second compilation unit allows the enumerated constants to be used as global values. That is, the constants can be referenced, but they cannot be initialized.

For more information about the enumerated type, see Chapter 7.

8.7 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 these data-type modifiers in detail.

8.7.1 The const Modifier

The **const** data-type modifier restricts access to stored data. If you declare an object to be of type **const**, you cannot modify that object.

The following rules apply to the use of the **const** data-type modifier:

- You can specify **const** with any of the other data-type keywords in a declaration.
- If you specify **const** when declaring an aggregate, all the aggregate members are treated as objects of type **const**.
- You can specify **const** with **volatile**, or any of the storage-class specifiers or modifiers.
- If you try to access a **const** object using a pointer to an object not declared as **const**, the result is undefined.
- The address of a non-**const** object can be assigned to a pointer to a **const** object, 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 example, the following section of code declares the variable y to be a constant pointer to an integer because the **const** modifier appears after the asterisk:

```
int * const y;
```

In the 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. Examples 1 and 2 show invalid 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;    /* invalid example  */
```

Example 2

```
globaldef {"psect"} int * const y;    /* invalid example  */
```

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 automatically causes 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, or your program may generate unpredictable results.

8.7.2 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 see the object using that pointer.

8.8 Storage-Class Modifiers

The VAX C compiler can accept a storage-class specifier and a storage-class modifier in any order; usually, the modifier is placed after the specifier in the source code. For example:

```
extern  noshare  int  x;

   /*  Or, equivalently...  */

int  noshare  extern  x;
```

The following sections describe each of the VAX C specific storage class modifiers in detail.

8.8.1 The noshare Modifier

The **noshare** storage-class modifier assigns the attribute NOSHR to the program section of the variable. This storage-class modifier is relevant only when used with VMS shared images. It is retained in VAX C/ULTRIX for reasons of compatibility with VAX C/VMS and has no meaning in VAX C/ULTRIX.

The **noshare** modifier can be used with the storage-class specifiers **static**, **[extern]**, **globaldef**, and **globaldef**{"name"}.

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**.

8.8.2 The readonly Modifier

The **readonly** storage-class modifier, like the **const** data-type modifier, 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 **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 **readonly** modifier restricts access to data in the same manner as the **const** modifier. However, in the declaration of a pointer, the **readonly** modifier cannot appear between the asterisk and the pointer variable to which it applies.

The following example shows the similarity between the **const** and **readonly** modifiers. In both instances, the variable point represents a constant pointer to a nonconstant integer.

```
readonly int * point;

int * const point;
```

NOTE

For new program development, Digital recommends that you use the **const** modifier.

8.8.3 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 any of the following declarations:

```
int  _align( QUADWORD )  data;
int  _align( quadword )  data;
int  _align( 3 )  data;
```

When specifying the boundary of the data alignment, you can either use a predefined constant or you can specify an integer value that is a power of 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 9

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, just as the name preprocessor implies. In VAX C, these directives are executed in an early phase of the compiler.

If you plan to port programs to and from other C implementations, take care in choosing which preprocessor directives to use within your programs. See Section 9.2 for more information about conditional compilation.

The preprocessor directives are introduced by number signs (#) that must appear in column 1 of the source listing. This chapter discusses the following preprocessor directives:

- **#define** and **#undef**—Define macro substitutions and replacements
- **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, **#endif**, and the **defined** operator—Controls under which conditions segments of code are to be compiled or not
- **#include**—Includes source text from an external file
- **#line** and **#**—Specifies a new line number and file name at the terminal, not in the listing file
- **#pragma**—Performs an implementation-specific task

Preprocessor directives are independent of the usual scope rules; they remain in effect from their occurrence until the end of the compilation unit. For more information about the compilation unit, seeChapter 2.

9.1 Macro Definitions (#define and #undef)

The **#define** directive specifies a macro identifier and a token string. The token string is substituted for every subsequent occurrence of that identifier in the program text, unless it occurs inside a **char** constant, a comment, or a quoted string. You use the **#undef** directive to cancel a definition for a macro.

NOTE

Previous versions of this guide refer to these macros as tokens.

The syntax of the **#define** directive is as follows:

```
#define identifier token-string
#define identifier(identifier, ...) token-string
```

If you omit the token string, the identifier is deleted from the text to be processed by the compiler.

After a token string is substituted in the source file, the compiler rescans the source line from the beginning of the substituted text to determine whether the previously inserted text contains identifiers defined by other **#define** directives. If so, the identifiers are replaced by their currently specified token strings. Example 9–1 shows nested **#define** directives.

Example 9–1: Nested Substitution Directives

```
/*  Show multiple substitutions and listing format          */

#define  AUTHOR  james + LAST

main()
{
   int writer,james,michener,joyce;

#define LAST michener
   writer = AUTHOR;

#define LAST joyce
   writer = AUTHOR;
}
```

Compile Example 9–1 with the following command:

```
% vcc -v example.lis -V "show = intermediate" example.c 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 line is the final text that the compiler processes:

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

You may continue the **#define** directive 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.

You can continue comments within the definition line without the backslash /newline. In the following example, all 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 9.1.2), and cannot be separated from the identifier.

9.1.1 Constant Identifiers

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

```
#define  EOF  (-1)
```

The substitution text for this example is delimited with parentheses to avoid lexical ambiguities when text is substituted in the program. For example:

```
i = EOF;
```

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

9.1.2 Macro Parameters

Some macros include a list of parameters. These macro substitutions look like function calls. If you call a function, control passes from the program to the function object code at run time; if you reference a macro, source code is inserted into the program at compile time. The parameters are replaced by the corresponding arguments and the text is inserted into the program stream. The syntax of a macro definition is as follows:

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

The name, parml, parm2, and so forth are identifiers, and token-string is arbitrary text.

After the macro definition, all macro references in the source code with the following form are replaced by the token string from the directive, and any formal parameters that appear in the token string are replaced by the corresponding arguments from the reference. For example, argument argl replaces parameter parml, and so forth, as follows:

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

As shown in the syntax of the macro definition, the token string is optional. If you omit the token string 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 these nested references are limited to a depth of 64. The maximum number of parameters or arguments is also 64.

The **lowertoupper** macro is a good example of macro substitution. For example:

```
#define  lowertoupper(c)   ((c) - 'a' + 'A')
```

When you reference the **lowertoupper** macro, the compiler replaces the macro keyword and its parameter with the token string from the directive.

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 that are specified according to the rules for identifiers in the VAX C language.
- You can use spaces, tabs, and comments freely within a **#define** directive. In particular, they can appear anywhere that the delta symbol (Δ) appears in the following example:

  ```
  #Δ defineΔ name(Δ parm1Δ ,Δ parm2Δ )Δ \
  Δ token-stringΔ
  ```

- White space cannot appear between the name and the left parenthesis that introduces the parameter list. White space can appear inside the token string. Also, at least one space, tab, or comment must separate name from **define**. Comments can 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, newlines, and form feeds) can be used freely within a macro reference. In particular, they can appear anywhere that the delta symbol appears in the following example:

  ```
  Δ nameΔ (Δ arg1Δ ,Δ arg2Δ )
  ```

- Arguments consist of arbitrary text. Syntactically, they are not restricted to VAX C expressions. They can 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, replacing 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 can cause side effects. Function calls are another source of possible side effects. Suppose the **lowertoupper** macro is defined as follows:

```
#define lowertoupper(c) ((c) >= 'a' && (c) <= 'z' ? (c) & 0X5F : (c))
```

Suppose the **lowertoupper** macro is invoked as follows:

```
lowertoupper(p++)
```

When the argument p++ is substituted in the macro definition, the effect within the program stream is as follows:

```
((p++) >= 'a' && (p++) <= 'z' ? (p++) & 0X5F : (p++))
```

The result of this expression may not be what was intended—that is, it may not be the uppercase letter corresponding to the value p++. For this reason, specifying macro arguments that may cause side effects is not good programming practice. Even if you are aware of possible side effects, the token strings within macro definitions are easily changed, which changes the side effects without warning.

9.1.3 Listing Substituted Lines

You can specify optional values in the **vcc** command line to force the listing of all lines that have been modified by macro substitutions. The optional values are **expansion** and **intermediate**. If your **vcc** command line includes **-V show=expansion** (as in the following example), 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.

```
% vcc -v example.lis -V "SHOW=EXPANSION" filepath.c RETURN
```

When you specify the option **intermediate**, the compiler lists all intermediate substitutions with one substitution per line, as in the following command example:

```
% vcc -v example.lis -V"SHOW=INTERMEDIATE" filepath.c RETURN
```

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

Example 9–1 shows the effect of the **-V show=intermediate** option. For more information about the format of VAX C compiler listings, seeChapter 2.

9.1.4 Canceling Definitions (#undef)

The following directive cancels a previous definition of the identifier by **#define**:

```
#undef identifier
```

If no previous definition exists, a warning message is generated if you specify **-V standard=portable**.

9.2 Conditional Compilation (#if, #ifdef, #ifndef, #else, #elif, and #endif)

Six directives are available to control conditional compilation. They delimit blocks of statements that are compiled if a certain condition is true. You can nest these directives. The beginning of the block of statements is marked by one of three directives: **#if**, **#ifdef**, or **#ifndef**. Optionally, an alternative block of statements can be set aside with the **#else** or the **#elif** directives. The end of the block is marked by an **#endif** directive.

If the condition checked by **#if**, **#ifdef**, or **#ifndef** is true, VAX C ignores all lines between an **#else** (or **#elif**) and an **#endif** directive.

If the condition is false, the lines between the **#if**, **#ifdef**, or **#ifndef** and an **#else**, (or **#elif**) or **#endif** directive are ignored. The compiler flags ignored lines with the letter X in the compiler listing margin.

The **#if** directive has the following form:

```
#if constant-expression
```

This directive checks whether the constant expression is nonzero (true). The operands must be constants. The increment (++), decrement (– –), **sizeof**, pointer (*), address (&), and cast operators are not allowed in the constant expression.

The constant expression in an **#if** directive is subject to text replacement and can contain references to identifiers defined in previous **#define** directives. The replacement occurs before the expression is evaluated.

If an identifier used in the expression is not currently defined, the compiler treats the identifier as though it were the constant 0. A warning message is generated if **–V standard=portable** is specified.

The **#ifdef** directive has the following form:

```
#ifdef identifier
```

This directive checks whether the identifier was previously defined by a **#define** directive.

The **#ifndef** directive has the following form:

```
#ifndef identifier
```

This directive checks to see if the identifier is not defined or if it has been undefined by the **#undef** directive.

The **#else** directive has the following form:

```
#else
```

This directive delimits alternative source lines to be compiled if the condition tested for in the corresponding **#if**, **#ifdef**, or **#ifndef** directive is false. An **#else** directive is optional.

The **#elif** directive has the following form:

```
#elif constant-expression
```

The **#elif** 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 following form:

```
#endif
```

This directive ends the scope of the corresponding **#if**, **#ifdef**, or **#ifndef** directives.

9.2.1 The defined Operator

If you need to check to see if many macros are defined, you may want to use the special **defined** operator in a single use of the **#if** line. In this way, you can check for macro definitions in one concise line without having to use many **#ifdef** or **#ifndef** directives.

For example, you might want to check the following macros:

```
#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.

9.3 File Inclusion (#include)

The **#include** directive inserts external text into the macro stream delivered to the compiler. Often, global definitions for use with the system library interfaces are included in the program stream with the **#include** directive. The **#include** directives may be nested to a depth determined by the limit of the number of concurrent open files for the process. The VAX C compiler imposes no inherent limitation on the nesting level of inclusion.

The following sections describe the forms of the **#include** directive.

9.3.1 Inclusion Using Angle Brackets (<>)

The first form of the directive is as follows:

#include <file-path>

The identifier file-path must be a valid file path name. The compiler first searches for the file relative to any directories specified with the **–I** option on the **vcc** command line. The compiler searches the directories in the order that they are specified on the command line. If the file is not found in any of these directories, the compiler looks for the file in the directory /usr/include. If the file is found, it is included in the compilation. If it is not found, the compiler generates an error.

9.3.2 Inclusion Using Quotation Marks (" ")

The second form of the **#include** preprocessor directive is as follows:

```
#include "file-path"
```

The identifier file-path must be a valid file path name. The compiler first searches for the file relative to the directory in which the including source file was found. If it is not found there, the compiler next searches for the file relative to any directories specified with the **-I** option on the **vcc** command. The compiler searches the directories in the order that they are specified on the command line. If the file is not found in any of these directories, the compiler looks for the file in the directory /usr/include. If the file is found, it is included in the compilation. If it is not found, the compiler generates an error.

9.3.3 Macro Substitution in #include Directives

VAX C allows macro substitution within the **#include** preprocessor directive.

For instance, if you want to include a file name, you can combine the **#define** and **#include** directives, as shown by the following example:

```
#define  token1  "file.ext"

#include token1
```

If you use defined macros in **#include** directives, the macros must evaluate to one of the two following acceptable **#include** file specifications or the use generates an error message:

```
<file-spec>
"file-spec"
```

9.4 Specifying Line Numbers (#line and #)

The VAX C compiler keeps track of information about relative line numbers in each file involved in the compilation and uses the number when it delivers diagnostic messages to the terminal. The compiler increments the subsequent lines from the line number specified by the **#line** directive. The directive can also specify a new file specification for the program source file. The **#line** directive does not change the line numbers in your compilation listing, only the line numbers given in 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 as follows:

```
#line constant identifier
#line constant string
# constant identifier
# constant string
```

The compiler gives the line following a **#line** directive the number specified by the parameter constant. You can specify the second parameter as either a VAX C identifier or a character-string constant. It supplies the valid file names. The character string must not exceed 255 characters.

9.5 Implementation-Specific Preprocessor Directive (#pragma)

This section describes the implementation-specific preprocessor directives, or pragmas, that are available in the VAX C compiler. The **#pragma** directive is a standard method for implementing features that vary from one compiler to the next.

Note that **#pragma** directives are subject to macro expansion. A macro reference can occur anywhere after the keyword **pragma**. The following example demonstrates this feature using the **#pragma inline** directive:

```
#define opt inline
#define f func
#pragma opt(f)
```

The **#pragma** directive becomes #pragma inline (func) after both macros are expanded.

The following sections describe the **#pragma** directives.

9.5.1 #pragma [no]builtins Directive

The **#pragma [no]builtins** directive disables or provides access to the VAX C predefined functions. These functions do not result in a reference to a function in the run-time library or in your program. Instead, the compiler generates the machine instructions necessary to carry out the function directly at the call site. (For information on available built-in functions, see Chapter 10.)

The **#pragma [no]builtins** directive has the following format:

```
#pragma builtins
#pragma nobuiltins
```

9.5.2 #pragma [no]inline Directive

The preprocessor directive **#pragma inline** suggests to the compiler that it provide inline expansion of the specified functions. Inline expansion of functions reduces execution time by replacing the function call with code that performs the actions of the original function code.

By default, VAX C attempts to provide inline expansion for all functions. The compiler also uses the following function characteristics to determine if it can provide inline expansion:

- Size
- Number of times the function is called
- Absence of the restrictions described in Section 9.5.2.1

The **#pragma inline** directive requests that the compiler attempt to provide inline code regardless of the size or number of times the specified functions are called. Functions that contain one of the restrictions described in Section 9.5.2.1 are never expanded inline, regardless of the use of the **#pragma inline**.

The **#pragma inline** directive has the following format:

```
#pragma inline (id, . . . )
```

id
Is a C function identifier.

For instance, the following example specifies that the functions push and pop be expanded inline throughout the module in which the **#pragma inline** appears:

```
void push( int );
int pop(void);

#pragma inline( push, pop)

int stack[100];
int *stackp = &stack;

void push(int x)
{
    if (stackp == &stack)
        *stackp = x;
    else
        *stackp++ = x;
}

int pop()
{
    return *stackp--;
}

main()
{
  push(1);
  printf("The top of stack is now %d \n",pop());
}
```

The **-V**"OPTIMIZE=NOINLINE" and the **-V**"NOOPTIMIZE" options disable all **#pragma inline** directives that appear in your source code.

The **#pragma noline** can be used selectively to identify functions that are not to be expanded inline, even when the **-V**"OPTIMIZE=INLINE" option is used on the **vcc** command line. The **#pragma noinline** directive has the following format:

#pragma noinline (id, . . .)

id
Is a C function identifier.

9.5.2.1 Restrictions on Inline Expansion

If a function is to be expanded inline, you must place the function definition in the same module as the function call. The definition can appear either before or after the function call.

Functions cannot be expanded inline if they perform the following tasks:

- Take the address of an argument.
- Use an index expression that is not a compile-time constant in an array that is a field of a **struct** argument. An argument that is a pointer to a **struct** is not restricted.
- Use the *varargs* package to access the function's arguments because they require arguments to be in adjacent memory locations, and inline expansion may violate that requirement.

When automatic inline expansion is not possible, no error or warning message is produced. When you explicitly request inline expansion by using the **#pragma inline** directive, a warning message is produced if inline expansion cannot be done.

9.5.3 #pragma [no]member_alignment Directive

By default, VAX C/ULTRIX aligns structure members on their natural boundaries. However, you can use **#pragma nomember_alignment** to explicitly specify member alignment on byte boundaries.

The **#pragma member_alignment** directive has the following format:

```
#pragma [no]member_alignment
```

When **#pragma member_alignment** is used (or defaulted), the compiler aligns structure members on the next boundary appropriate to the type of the member, rather than on the next byte. For instance, a **long** variable is aligned on the next longword boundary; a **short** variable is aligned on the next word boundary.

Consider the following example:

```
#pragma nomember_alignment

struct x {
           char c;
           int b;
           };

#pragma member_alignment

struct y {
          char c;       /*3 bytes of filler follow c */
          int b;
          };

main ()

{
        printf( "The sizeof y is: %d\n", sizeof (struct y) );
        printf( "The sizeof x is: %d\n", sizeof (struct x) );
}
```

When this example is executed, it shows the difference between **#pragma member_alignment** and the directive **#pragma nomember_alignment**. The difference can also be seen by compiling **–V** x.lis **–V show=symbols** and comparing the listed information for the two structures.

Once used, the **nomember_alignment** pragma remains in effect until the **member_alignment** pragma is encountered.

9.5.4 #pragma [no]standard Directive

Use **#pragma nostandard** to tell VAX C to ignore the current setting of the **–V standard=portable** option until further notice. It has no effect if the qualifier is not specified.

The **#pragma nostandard** directive has the following format:

```
#pragma nostandard
```

Use **#pragma standard** to tell VAX C to reinstate the setting of the **–V standard=portable** option. This pragma does not turn on portability checking if the **–V standard=portable** option is not specified on the **vcc** command line.

The **#pragma standard** directive has the following format:

```
#pragma standard
```

The **nostandard** and **standard** pragmas are together to define regions of source code where portability diagnostics are never to be issued. The following example demonstrates the use of these pragmas:

```
#pragma nostandard
extern noshare FILE *stdin, *stdout, *stderr;
#pragma standard
```

In this example, **nostandard** prevents the NONPORTCLASS diagnostic from being issued against the **noshare** storage-class modifier, which is VAX C specific.

Chapter 10

Predefined Macros and Built-In Functions

This chapter describes the following topics:

- Predefined macros
- Built-in functions

VAX C predefines these macros and functions for your programming convenience. The macros assist in transporting code and performing simple tasks that are common to many programs. The built-in functions access VAX instructions very efficiently.

10.1 Predefined Macros

The following sections describe the VAX C predefined macros for use in your programs.

10.1.1 System-Identification Macros

VAX C automatically defines macros that can be used to identify the system on which the program is running. These macros can assist in writing code that executes conditionally, depending on whether the program is running on a Digital system or some other system. These symbols are defined as if the following text fragment were included by the compiler before every compilation source group:

```
#define bsd4_2         1
#define ultrix         1
#define unix           1
#define vax            1
#define VAX            1
#define vaxc           1
#define VAXC           1
#define vax11c         1
#define VAX11C         1
```

You can use these definitions to separate portable and nonportable code in any of your VAX C programs.

You can use these symbols to conditionally compile VAX C programs used on more than one operating system to take advantage of system-specific features. See Section 9.2 for more information about using the preprocessor conditional compilation directives.

Consider the following example:

```
#if       VAXC
#include  <descript.h>       /* Include descriptor definitions   */
#endif
```

10.1.2 CC$gfloat (G_Floating Identification Macro)

VAX C automatically defines a macro that can be used to identify whether you are compiling your program using the G_floating option. This macro can assist in writing code that executes conditionally, depending on whether the program is running using D_floating or G_floating precision.

If you compile your program using the **–Mg** option, this symbol is defined as if the following were included before every compilation source group:

```
#define  CC$gfloat  1
```

If you did not compile your program using the **–Mg** option, this symbol is defined as if the following were included before every compilation source group:

```
#define  CC$gfloat  0
```

You can conditionally assign values to variables of type **double** without causing an error and without being certain of how much storage was allocated for the variable. For example, external variables may be assigned values as follows:

```
#if CC$gfloat
double x = 0.12e308;        /* Range to 10 to the 308th power */
#else
double x = 0.12e38;         /* Range to 10 to the 38th power  */
#endif
```

The VAX C compiler determines whether or not to substitute the value 1 for every occurrence of the predefined identifiers in a program; these identifiers are reserved by Digital. The effect of these definitions may be removed by explicitly undefining the conflicting name. See Section 9.1.4 for more information about undefining. For more information about the G_floating representation of the **double** data type, see Chapter 7.

10.1.3 The __DATE__ Macro

The __DATE__ macro evaluates to a string specifying the date on which the compilation started. The string presents the date in the following format:

Mmm-dd-yyyy

The first d is a space if dd is less than 10.

The following is an example of the __DATE__ macro:

```
printf("%s",_ _DATE_ _);
```

10.1.4 The __FILE__ Macro

The __FILE__ macro evaluates to a string specifying the file specification of the current source file. The following is an example of the __FILE__ macro:

```
printf("file %s" _ _FILE_ _);
```

10.1.5 The __LINE__ Macro

The __LINE__ macro evaluates to an integer specifying the number of the line in the source file containing the macro reference. The following is an example of the __LINE__ macro:

```
printf("At line %d in file %s", _ _LINE_ _, _ _FILE_ _);
```

10.1.6 The __TIME__ Macro

The __TIME__ macro evaluates to a string specifying when the compilation started. The string presents the time in the following format:

hh:mm:ss

The following is an example of the __TIME__ macro:

```
printf("%s", _ _TIME_ _);
```

10.2 Built-In Functions

The following sections describe the built-in functions that allow you to directly access the VAX hardware and machine instructions to perform operations that are cumbersome, slow, or impossible in pure C.

These functions are very efficient because they are built into the VAX C compiler. This means that a call to one of these functions does not result in a reference to a function in the run-time library or to a function in your program. Instead, the compiler generates the machine instructions necessary to carry out the function directly at the call site. Because most of these built-in functions closely correspond to single VAX machine instructions, the result is small, fast code.

Some of these built-in functions (such as those that operate on strings or bits) are of general interest. Others (such as the functions dealing with process context) are of interest if you are writing device drivers or other privileged software. Some of the functions discussed in the following sections are privileged and unavailable to user mode programs.

You must place the following pragma in your source file before using one or more built-in functions:

```
#pragma builtins
```

Some of the built-in functions have optional arguments or allow a particular argument to have one of many different types. To describe the different legal combinations of arguments, the description of each built-in function may list several different prototypes for the function. As long as a call to a built-in function matches one of the prototypes listed, the call is legal. Furthermore, any legal call to a built-in function acts as if the corresponding prototype were in scope. Thus, the compiler performs the argument checking and argument conversions specified by that prototype.

The majority of the built-in functions are named after the VAX instruction that they generate. The built-in functions provide direct and unencumbered access to those VAX instructions. Any inherent limitations to those instructions are limitations to the built-in functions as well. For instance, the MOVC3 instruction and the _MOVC3 built-in function can move at most 65,535 characters.

10.2.1 Add Aligned Word Interlocked (_ADAWI)

The _ADAWI function adds its source operand to the destination. This function is interlocked against similar operations by other processors or devices in the system.

The _ADAWI function has the following formats:

```
int _ADAWI(short src, short *dest);
int _ADAWI(short src, unsigned short *dest);
```

src
Is the value to be added to the destination.

dest
Is a pointer to the destination. The destination must be aligned on a word boundary. (One way to achieve alignment is to use **_align**.)

There are three possible return values, as follows:

- –1, if the sum when considered to be a signed number is negative
- 0, if the sum is zero
- 1, if the sum is positive

10.2.2 Branch on Bit Clear-Clear Interlocked (_BBCCI)

The _BBCCI function performs the following functions in interlocked fashion:

- Returns the complement of the bit specified by the two arguments
- Clears the bit specified by the two arguments

The _BBCCI function has the following format:

```
int _BBCCI(int position, void *address);
```

position
Is the position of the bit within the field.

address
Is the base address of the field.

The return value is 0 or 1, which is the complement of the value of the specified bit before being cleared.

10.2.3 Branch on Bit Set-Set Interlocked (_BBSSI)

The _BBSSI function performs the following functions in interlocked fashion:

- Returns the status of the bit specified by the two arguments
- Sets the bit specified by the two arguments

The _BBSSI function has the following format:

```
int _BBSSI(int position, void *address);
```

position
Is the position of the bit within the field.

address
Is the base address of the field.

The return value is 0 or 1, which is the value of the specified bit before being set.

10.2.4 Find First Clear Bit (_FFC)

The _FFC function finds the position of the first clear bit in a field. The bits are tested for clear status starting at bit 0 and extending to the highest bit in the field.

The _FFC function has the following format:

```
int _FFC(int start, char size, const void *base, int *position);
```

start
Is the start position of the field.

size
Is the size of the field, in bits. The size must be a value from 0 to 32 bits.

base
Is the address of the field.

position
Is the address of an integer to receive the position of the clear bit. If no bit is clear, the integer is set to the position of the first bit to the left of the last bit tested.

There are two possible return values, as follows:

- 0, if all bits in the field are set
- 1, if a bit with value 0 is found

10.2.5 Find First Set Bit (_FFS)

The _FFS function finds the position of the first set bit in a field. The bits are tested for set status starting at bit 0 and extending to the highest bit in the field.

The _FFS function has the following format:

```
int _FFS(int start, char size, const void *base, int *position);
```

start
Is the start position of the field.

size
Is the size of the field, in bits. The size must be a value from 0 to 32 bits.

base
Is the address of the field.

position
Is the address of an **int** to receive the position of the set bit. If no bit is set, the integer is set to the position of the first bit to the left of the last bit tested.

There are two possible return values, as follows:

- 0, if all bits in the field are clear
- 1, if a bit with value 1 is found

10.2.6 Halt (_HALT)

The _HALT function halts the processor when executed by a process running in kernel mode. This is a privileged function.

The _HALT function has the following format:

```
void _HALT(void);
```

10.2.7 Insert Entry into Queue at Head Interlocked (_INSQHI)

The _INSQHI function inserts an entry into the front of a queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _INSQHI function has the following format:

```
int _INSQHI(void *new_entry, void *head);
```

new_entry
Is a pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

head
Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

There are three possible return values, as follows:

- 0, if the entry was inserted, but it was not the only entry in the list
- 1, if the entry was not inserted because the secondary interlock failed
- 2, if the entry was inserted and it was the only entry in the list

10.2.8 Insert Entry into Queue at Tail Interlocked (_INSQTI)

The _INSQTI function inserts an entry at the end of a queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _INSQTI function has the following format:

```
int _INSQTI(void *new_entry, void *head);
```

new_entry
Is a pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

head
Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

There are three possible return values, as follows:

- 0, if the entry was inserted, but it was not the only entry in the list
- 1, if the entry was not inserted because the secondary interlock failed
- 2, if the entry was inserted and it was the only entry in the list

10.2.9 Insert Entry in Queue (_INSQUE)

The _INSQUE function inserts a new entry into a queue following an existing entry.

The _INSQUE function has the following format:

```
int _INSQUE(void *new_entry, void *predecessor);
```

new_entry
Is a pointer to the new entry to be inserted.

predecessor
Is a pointer to an existing entry in the queue.

There are two possible return values, as follows:

- 0, if the entry was the only entry in the queue
- 1, if the entry was not the only entry in the queue

10.2.10 Load Process Context (_LDPCTX)

The _LDPCTX function restores the register and memory-management context. This is a privileged function.

The _LDPCTX function has the following format:

```
void _LDPCTX(void);
```

10.2.11 Locate Character (_LOCC)

The _LOCC function locates the first character in a string matching the target character.

The _LOCC function has the following formats:

```
int _LOCC(char target, unsigned short length,
          const char *string);
```

```
int _LOCC(char target, unsigned short length,
          const char *string, char **position);
```

target
Is the character being searched.

length
Is the length of the searched string. The length must be a value from 0 to 65,535.

string
Is a pointer to the searched string.

position
Is a pointer to a pointer to a character. If the searched character is found, the pointer pointed to by position is updated to point to the character found. If the character is not found, the pointer pointed to by position is set to the address one byte beyond the string. This is an optional argument.

If the target character is found, the return value is the number of bytes remaining in the string; otherwise, the return value is 0.

10.2.12 Move from Processor Register (_MFPR)

The _MFPR function returns the contents of a processor register. This is a privileged function.

The _MFPR function has the following formats:

```
void _MFPR(int register_num, int *destination);
void _MFPR(int register_num, unsigned int *destination);
```

register_num
Is the number of the privileged register to be read.

destination
Is a pointer to the location receiving the value from the register. This location may be a **signed** or **unsigned int**.

10.2.13 Move Character 3 Operand (_MOVC3)

The _MOVC3 function copies a block of memory. It is the preferred way to copy a block of memory to a new location.

The _MOVC3 function has the following formats:

```
void _MOVC3(unsigned short length, const char *src, char *dest);

void _MOVC3(unsigned short length, const char *src, char *dest,
            char **endsrc);

void _MOVC3(unsigned short length, const char *src, char *dest,
            char **endsrc, char **enddest);
```

length
Is the length of the source string, in bytes. The length must be a value from 0 to 65,535.

src
Is a pointer to the source string.

dest
Is a pointer to the destination memory.

endsrc
Is a pointer to a pointer. The _MOVC3 function sets the pointer that is pointed to by endsrc pointing to the address of the byte beyond the source string. It is optional if the enddest argument is not given.

enddest
Is a pointer to a pointer. The _MOVC3 function sets the pointer pointed to by endsrc to the address of the byte beyond the destination string. This is an optional argument.

10.2.14 Move Character 5 Operand (_MOVC5)

The _MOVC5 function allows the source string specified by the pointer and length pair to be moved to the destination string specified by the other pointer and length pair. If the source string is smaller than the destination string, the destination string is padded with the specified character.

The _MOVC5 function has the following formats:

```
void _MOVC5(unsigned short srclen, const char *src, char fill,
              unsigned short destlen, char *dest);
```

```
void _MOVC5(unsigned short srclen, const char *src, char fill,
              unsigned short destlen, char *dest,
        unsigned short *unmoved_src);
```

```
void _MOVC5(unsigned short srclen, const char *src, char fill,
              unsigned short destlen, char *dest,
        unsigned short *unmoved_src, char **endsrc);
```

```
void _MOVC5(unsigned short srclen, const char *src, char fill,
              unsigned short destlen, char *dest,
        unsigned short *unmoved_src, char **endsrc,
              char **enddest);
```

srclen
Is the length of the source string, in bytes. The length must be a value from 0 to 65,535.

src
Is a pointer to the source string.

fill
Is the fill character to be used if the source string is smaller than the destination string.

destlen
Is the length of the destination string, in bytes. The length must be a value from 0 to 65,535.

dest
Is a pointer to the destination string.

unmoved_src
Is a pointer to a short integer that the _MOVC5 function sets to the number of unmoved bytes remaining in the source string.

endsrc
Is a pointer to a pointer. The _MOVC5 function sets the pointer pointed to by endsrc pointing to the address of the byte beyond the source string. It is optional if the enddest argument is not given.

enddest
Is a pointer to a pointer. The _MOVC5 function sets the pointer pointed to by endsrc to the address of the byte beyond the destination string. This is an optional argument.

10.2.15 Move from Processor Status Longword (_MOVPSL)

The _MOVPSL function stores the value of the Processor Status Longword (PSL).

The _MOVPSL function has the following formats:

```
void _MOVPSL(int *psl);
void _MOVPSL(unsigned int *psl);
```

psl
Is the address of the location for storing the value of the Processor Status Longword.

10.2.16 Move to Processor Register (_MTPR)

The _MTPR function loads a value into one of the special processor registers. It is a privileged function.

The _MTPR function has the following format:

```
int _MTPR(int src, int register_num);
```

src
Is the value to store into the processor register.

register_num
Is the number of a privileged register to be updated.

The return value is the V condition flag from the Processor Status Longword (PSL).

10.2.17 Probe Read Accessibility (_PROBER)

The _PROBER function checks to see if you can read the first and last byte of the given address and length pair.

The _PROBER function has the following format:

```
int _PROBER(char mode, unsigned short length, const void *address);
```

mode
Is the processor mode used for checking the access.

length
Is the length of the memory segment, in bytes. The length must be a value from 0 to 65,535.

address
Is the pointer to the memory segment to be tested for read access.

There are two possible return values, as follows:

- 0, if both bytes are not accessible
- 1, if both bytes are accessible

10.2.18 Probe Write Accessibility (_PROBEW)

The _PROBEW function checks the write accessibility of the first and last byte of the given address and length pair.

The _PROBEW function has the following format:

```
int _PROBEW(char mode, unsigned short length, const void *address);
```

mode
Is the processor mode used for checking the access.

length
Is the length of the memory segment, in bytes. The length must be a value from 0 to 65,535.

address
Is the pointer to the memory segment to be tested for write access.

There are two possible return values, as follows:

- 0, if both bytes are not accessible
- 1, if both bytes are accessible

10.2.19 Read General-Purpose Register (_READ_GPR)

The _READ_GPR function returns the value of a general-purpose register.

The _READ_GPR function has the following format:

```
int _READ_GPR(int register_num);
```

register_num
Is an integer constant expression giving the number of the general-purpose register to be read.

The return value is the value of the general-purpose register.

10.2.20 Remove Entry from Queue at Head Interlocked (_REMQHI)

The _REMQHI function removes the first entry from the queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _REMQHI function has the following format:

```
int _REMQHI(void *head, void **removed_entry);
```

head
Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

removed_entry
Is a pointer to a pointer that _REMQHI sets to point to the removed entry.

There are four possible return values, as follows:

- 0, if the entry was removed and the queue has remaining entries
- 1, if the entry could not be removed because the secondary interlock failed
- 2, if the entry was removed and the queue is now empty
- 3, if the queue was empty

10.2.21 Remove Entry from Queue at Tail Interlocked (_REMQTI)

The _REMQTI function removes the last entry from the queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _REMQTI function has the following format:

```
int _REMQTI(void *head, void **removed_entry);
```

head
Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

removed_entry
Is a pointer to a pointer that _REMQTI sets to point to the removed entry.

There are four possible return values, as follows:

- 0, if the entry was removed and the queue has remaining entries
- 1, if the entry could not be removed because the secondary interlock failed
- 2, if the entry was removed and the queue is now empty
- 3, if the queue was empty

10.2.22 Remove Entry from Queue (_REMQUE)

The _REMQUE function removes an entry from a queue.

The _REMQUE function has the following format:

```
int _REMQUE(void *entry, void **removed_entry);
```

entry
Is a pointer to the queue entry to be removed.

removed_entry
Is a pointer to a pointer that _REMQUE sets to the address of the entry removed from the queue.

There are three possible return values, as follows:

- 0, if the entry was removed and the queue has remaining entries

- 1, if the entry was removed and the queue is now empty
- 2, if the queue was empty

10.2.23 Scan Characters (_SCANC)

The _SCANC function locates the first character in a string with the desired attributes. The attributes are specified through a table and a mask.

The _SCANC function has the following formats:

```
int _SCANC(unsigned short length, const char *string,
           const char *table, char mask);
```

```
int _SCANC(unsigned short length, const char *string,
           const char *table, char mask, char **match);
```

length
Is the length of the string to scan, in bytes. The length must be a value from 0 to 65,535.

string
Is a pointer to the string to scan.

table
Is a pointer to the table.

mask
Is the mask.

match
Is a pointer to a pointer that the _SCANC function sets to the address of the byte that matched. (If no match occurs, it is set to the address of the byte following the string.) This is an optional argument.

The return value is the number of bytes remaining in the string if a match was found; otherwise, the return value is 0.

10.2.24 Simple Read (_SIMPLE_READ)

The _SIMPLE_READ function reads I/O registers or shared memory. It causes a MOVB, MOVW, or MOVL instruction to be generated that cannot be moved or modified during optimization.

The _SIMPLE_READ function has the following formats:

```
char _SIMPLE_READ(const char *source);
short _SIMPLE_READ(const short *source);
int _SIMPLE_READ(const int *source);
long _SIMPLE_READ(const long *source);
```

source
Is a pointer to the source to be read. The object being pointed to must be a signed integer. The type of the object pointed to determines the type of the function result.

The return value is the value of the specified source.

10.2.25 Simple Write (_SIMPLE_WRITE)

The _SIMPLE_WRITE function writes to I/O registers or shared memory. It causes a MOVB, MOVW, or MOVL instruction to be generated that cannot be moved or modified during optimization.

The _SIMPLE_WRITE function has the following formats:

```
void _SIMPLE_WRITE(char value, char *dest);
void _SIMPLE_WRITE(short value, short *dest);
void _SIMPLE_WRITE(int value, int *dest);
void _SIMPLE_WRITE(long value, long *dest);
```

value
Is the value to be stored. The type of the destination argument determines the type of this argument.

dest
Is a pointer to the destination. The type of the object pointed to by dest must be a signed integer type. The type of this object determines the type of the first argument to this function.

10.2.26 Skip Character (_SKPC)

The _SKPC function locates the first character in a string that does not match the target character.

The _SKPC function has the following formats:

```
int _SKPC(char target, unsigned short length, const char *string);

int _SKPC(char target, unsigned short length, const char *string,
          char **position);
```

target
Is the target character.

length
Is the length of the string, in bytes. The length must be a value from 0 to 65,535.

string
Is a pointer to the string to scan.

position
Is a pointer to a pointer. The _SKPC function sets the pointer pointed to by position to the address of the nonmatching character. (If all the characters match, it is set to the address of the first byte beyond the string.) This is an optional argument.

The return value is the number of bytes remaining in the string if an unequal byte was located; otherwise, the return value is 0.

10.2.27 Span Characters (_SPANC)

The _SPANC function locates the first character in a string without certain attributes. The attributes are specified through a table and a mask.

The _SPANC function has the following formats:

```
int _SPANC(unsigned short length, const char *string,
           const char *table, char mask);
```

```
int _SPANC(unsigned short length, const char *string,
           const char *table, char mask, char **position);
```

length
Is the length of the string, in bytes. The length must be a value from 0 to 65,535.

string
Is a pointer. It points to the string to be scanned.

table
Is a pointer to the table.

mask
Is the mask.

position
Is a pointer to a pointer. The _SPANC function sets the pointer pointed to by position to the address of the byte that does not match the attributes. (If all the characters in the string match, this pointer is set to the address of the first byte beyond the string.) This is an optional argument.

The return value is the number of bytes remaining in the string if a match was found; otherwise, the return value is 0.

10.2.28 Save Process Context (_SVPCTX)

The _SVPCTX function saves the context of a process. The general-purpose registers are saved in the process control block, which is later used to resume a process. This function is privileged.

The _SVPCTX function has the following format:

```
void _SVPCTX(void);
```

10.2.29 Write General-Purpose Register (_WRITE_GPR)

The _WRITE_GPR function loads a value into a specified general-purpose register.

The _WRITE_GPR function has the following format:

```
void _WRITE_GPR(int value, int register_num);
```

value
Is the value to load into the register.

register_num
Is an integer constant expression giving the number of the general-purpose register to be loaded. The register number must be a value from 0 to 15.

Appendix A
The lk Linker

This appendix describes the command-line interface to the lk linker. The lk linker is invoked when you specify the **–V lk_object** option on the **vcc** command line.

You must invoke the lk linker to link files produced by a version of vcc prior to Version 4.0 or files with a .obj extension. The default ld linker cannot link these files.

The lk linker, like the ld linker, is a linkage editor utility. It takes object modules as input and builds an executable image as output. The main purpose of both linkers is to resolve external references between modules in the program.

The output from the lk linker is a standard ULTRIX a.out object module.

A.1 The lk Command Line

The **lk** command line has the following format:

lk [*–option* [*option-args*]]... *file...*

***–option* [*option-args*]**
The **lk** command options are almost identical to the options accepted by the **ld** command, and both linkers process the options in the same fashion (that is, the individual options have the same effects). The only differences are that the **ld** command options **–A**, **–d**, and **–r** are not supported by the **lk** command, and the **lk** command option **–K** is not supported by the **ld** command.

NOTE

> All **lk** command options can be specified with the **vcc** command.

Table A–1 describes the options for the **lk** command.

file
Files specified on the **lk** command line may be object files or library files. If a file is an object file (identified by the OMAGIC, NMAGIC, or ZMAGIC number at the appropriate place in the file), the linker processes the file's contents and concatenates the output to the program that it is building. Thus, the order in which object files are presented to the linker determines the order in which routines appear in the resulting program.

If the file is an archive library (identified by the presence of ARMAGIC at the appropriate place in the file), the linker searches the files in the library sequentially once for unresolved external symbols. If a resolution is found, that file is incorporated in the program being built and the search continues (possibly with new external symbols from the newly loaded file). If the library contains a ranlib-built dictionary file (_SYMDEF), the linker searches the dictionary iteratively to satisfy as many external references as possible instead of sequentially searching the archive.

Table A–1: Command Options Supported by the lk Linker

Command Option	Description
–A pathname	Puts incremental loading in effect. Linking is to be done so that the resulting object can be read into a program that is executing. The argument is the name of a file whose symbol table will be used to define additional symbols. Only newly linked material is entered into the text and data portions of a.out, but the new symbol table reflects every symbol defined before and after the incremental load. This argument must appear before any other object file in the argument list. You can use the **–T** option as well, which means that the newly linked segment will begin at the corresponding address (which must be a multiple of 1024). The default value is the old value of _end.
–D number	Sets the data segment length. Number is a number specifying the desired length of the data segment. The linker pads the data segment with 0 bytes to this length.
–e symbol	Sets the entry point. Symbol is the name of the entry point for the program. The default is location 0.
–H number	Sets the start of the data section. The linker adds the specified number to the end of the text address and causes the data section to start at that address.
–lx	Specifies a search library. The linker searches library /lib/libx.a (where x is the string specified as an argument to the **–l** option) for unresolved symbols. If this library does not exist, the linker searches /usr/lib/libx.a. If unresolved symbols remain, the linker tries /usr/local /lib/libx.a. The positioning of the **–l** option on the command line is significant: library searching occurs at the point in the link where the option occurs.
–K	Produces a full load map. The load map lists the files and modules included in the program, the allocation of psects to memory, and a cross-reference of all symbols (by name and by value). The map file is name.map, where name is the file name of the output file (as specified by the **–o** option). See Chapter 2 for a sample listing of a map file.
–M	Produces a short load map. **–M** is the same as **–K**, except that the symbolic cross-reference includes only those symbols that were referenced.
–N	The linker produces a file in OMAGIC format. The text portion of the file will be read/write and not shared.
–n	The linker produces a file in NMAGIC format. The text portion of the file will be read-only and shared among all users executing the file. The data segment starts at the first possible 1024-byte boundary following the end of the text segment.
–o file	Sets the output file name. The linker uses file as the name for the output file. The default is a.out.
–S	Removes some symbols. The linker does not include any symbols in the a.out file except for locals and globals.
–s	Removes all symbols and relocation information. The linker does not include any symbols or relocation information in the a.out file.

(continued on next page)

Table A–1 (Cont.): Command Options Supported by the lk Linker

Command Option	Description
–T number	Sets the text segment origin. The linker offsets the beginning of the text segment at the address given by the hexadecimal number specified as number. The default is 0.
–t	Traces. The linker writes the name of each file to standard output as it is processed.
–u symbol	Sets an undefined symbol. The linker adds the symbol name specified by symbol as an undefined symbol in the symbol table. This option is useful when loading entirely from libraries, since the symbol table is empty initially and an unresolved reference is needed to force the loading of the first routine.
–X	Removes symbols starting with L. The linker does not include symbols starting with L in the symbol table of the a.out file. Some compilers use such names for internally generated labels. The **–X** option provides a way to eliminate just these names.
–x	Removes local symbols. The linker includes only global symbols in the symbol table of the a.out file.
–Y [option]	Compile file for one of the following options: SYSTEM_FIVE, BSD, or POSIX.
–y symbol	Traces a symbol. The linker indicates each file that the specified symbol appears in and whether the file defines or references it. The **–y** option may occur many times to trace several symbols.
–z	The linker generates a.out in ZMAGIC format. The program is loaded on demand instead of being preloaded. This is the default format. The a.out header is 1024 bytes long. The text and data segments are padded with 0 bytes to the next higher 1024-byte boundary.

A.2 Linker Processing

The lk linker processes two different kinds of entities when building the executable program image: modules and program sections (psects).

A module is a unit of program compilation. There is usually one module per object file or library element. Some language translators, such as the as assembler, do not generate explicit modules. The linker assumes these translators to have one module consisting of the entire object file or library element.

A program section is a unit of virtual memory allocation. A program section specifies the attributes of the virtual memory to be allocated (for example, executable as opposed to data, read-only as opposed to read/write, absolute as opposed to relocatable, initialized as opposed to uninitialized). The linker uses the attributes of a program section to assign virtual memory addresses to the module contributions and to the symbols defined in the modules.

The psect also indicates the boundary alignment required for the segment of virtual memory. The data in an object module specifies, either explicitly or implicitly, the program section where the executable code and data in the module are to be allocated.

A.2.1 Program Section Attributes

This section describes each program section attribute and its effect on image processing by the lk linker. The attributes are grouped into mutually exclusive pairs as follows:

- Relocatable (REL) and Absolute (ABS)

  A relocatable program section is one that the linker can position in virtual memory according to the memory allocation strategy for the type of image being produced.

  The linker does not allocate virtual memory for an absolute program section. An absolute program section has no data or code, and appears as if it were based at virtual address 0. Absolute program sections are used primarily to define global symbols.

- Concatenated (CON) and Overlaid (OVR)

  These attributes govern how the linker allocates virtual memory to program section contributions from more than one module.

  If the program section has the concatenated attribute, the linker places each module's contribution in contiguous memory addresses. For example, if both modulea and moduleb contribute to program section psect1, and psect1 has the concatenated attribute, the linker allocates virtual memory for modulea's contribution and then allocates additional space for moduleb's contribution. Thus, the total size of a program section defined with the concatenated attribute is the sum of each module's contribution plus any padding allowed for individual alignments.

  If the program section has the overlaid attribute, the linker assigns each module's contribution the same base address, so that the contributions overlay each other. The total size of an overlaid program section is the size of the largest contribution. Any contribution to an overlaid program section can initialize the contents of the section. However, the final contents are determined by the last contributing module. Thus, the order in which you specify input modules is important.

  VAX FORTRAN common blocks and C external variables are implemented with overlaid program sections.

- Writeability (WRT and NOWRT)

  The writeability attribute determines whether the contents of the program section are protected against modification when you execute the program. Program sections with the nonwriteable attribute are allocated in the text section of the image. Program sections with the writeable attribute are allocated in the data or BSS sections, based on the modified attribute.

- Modified (MOD) and Unmodified (NOMOD)

  The modified attribute tells the linker whether any modules initialize the contents of the program section. Writeable, unmodified program sections are the only ones that the linker places in the BSS section of the image.

- Alignment

  The program section alignment attribute specifies the power-of-two boundary on which the linker is to align the program section code or data. The linker can align on any power of two in the range 0 (byte alignment) to 9 (512-byte boundary alignment).

- Local (LCL) and Global (GBL) Scope

If the global and overlaid attributes occur together, the program section is subject to special treatment from the linker for resolving references to .COMM symbols and to symbols with the same name as the program section. See Section A.2.3 for a description of how this is handled.

The following attributes are reserved for possible future implementation:

- Executability (EXE and NOEXE)
- Readability (RD and NORD)
- Position Independence (PIC and NOPIC)
- Shareability (SHR and NOSHR)
- User (USR) and Library (LIB)
- Protection (VEC and NOVEC)

A.2.2 Virtual Memory Allocation by the Linker

An ULTRIX executable program image file contains three virtual memory sections: a program text (executable code) section, an initialized data section, and a BSS (uninitialized data) section. The linker allocates virtual memory to the sections in that order: text, initialized data, BSS. The linker orders program sections in each image virtual memory section according to the program section attributes, as follows:

Image Section	Program Section Attributes			
text	NOWRT	EXE	NOVEC	—
	NOWRT	EXE	VEC	—
	NOWRT	NOEXE	NOVEC	—
	NOWRT	NOEXE	VEC	—
data	WRT	EXE	NOVEC	MOD
	WRT	EXE	VEC	MOD
	WRT	NOEXE	NOVEC	MOD
	WRT	NOEXE	VEC	MOD
BSS	WRT	—	—	NOMOD

Within an attribute group, the linker allocates program sections in alphabetical program section name order, with one exception: program section ULT$TEXT always occurs first in the text section.

Program sections with the global and overlaid attributes are handled in a special way. The linker constructs a new name by converting the program section name to lowercase and prefixing an underscore to it. If this name matches a symbol defined in a program section with different attributes (that is, other than GBL,OVR), the linker bases the global, overlaid program section at the virtual address given by the value of the symbol that was matched. This processing allows global, overlaid program sections to be initialized from a.out-format modules created by the as assembler, VAX C compiler, portable C compiler, portable Pascal compiler, and f77 compiler.

The virtual memory base address for the program image sections themselves depends on the type of image being generated as follows:

- Read/Write Text (OMAGIC format)

  The data section immediately follows the end of the text section, plus any padding specified by the **–H** command option. The BSS section immediately follows the end of the data section, plus any padding specified by the **–D** command option.

- Read-Only Text (NMAGIC format)

  The data section starts on the next page (1024-byte) boundary following the end of the text section, plus any padding from the **–H** command option. The BSS section immediately follows the end of the data section, plus any padding specified by the **–D** command option.

- Demand Loadable (ZMAGIC format)

  The data section starts on the next page (1024-byte) boundary following the end of the text section, plus any padding from the **–H** command option. The data segment, plus any padding specified by the **–D** command option, is padded to the next highest page boundary. Thus, the BSS section starts on the next page boundary following the end of the data section plus any padding specified by the **–D** command option. However, the linker will allocate program sections with the modified attribute starting at the end of the data section (plus **–D** value) to avoid wasting the space resulting from data segment roundup.

A.2.3 Special Processing for Modules Produced by the ld Linker

Modules in a.out(5) format, the format generated by the as assembler and processed by the ld linker, receive some special processing from the lk linker. These modules do not have any explicit module or program section declarations, and some symbols require special handling. The special rules for files and library elements with a.out(5) format are as follows:

- The linker considers the file or library element to be a single module with the same name as the file or library element.
- The module declares and makes contributions to three program sections. All program sections and contributions have alignment value 2 (longword alignment).

  1. The following program section receives the contents of the text section of the module:

     ```
     ULT$TEXT (PIC, USR, CON, REL, GBL, EXE, RD, NOWRT, NOVEC, MOD)
     ```

  2. The following program section receives the contents of the data section of the module:

     ```
     ULT$DATA (PIC, USR, CON, REL, GBL, NOEXE, RD, WRT, NOVEC, MOD)
     ```

  3. The following program section receives the contents of the BSS section of the module, and the symbols from the linker-generated module $$COMSYMS:

     ```
     ULT$COMM (PIC, USR, CON, REL, GBL, NOEXE, RD, WRT, NOVEC, NOMOD)
     ```

- The linker processes .COMM symbols from modules with a.out(5) format in a special way. These symbols represent C external variables, COMMON areas from the f77 compiler, and other such entities that the linker normally processes as program sections with the overlaid attribute.
  - If the name of a .COMM symbol matches a symbol that is defined elsewhere and is not a .COMM symbol, the .COMM symbol is treated as a reference to that symbol.
  - If the name of a .COMM symbol matches the name of a program section with the global and overlaid attributes (after the program section name is converted to lowercase and prefixed with an underscore), the linker treats the .COMM symbol as a reference to offset 0 in that program section.
  - If neither of the previous two conditions apply, the linker increases the size of program section $$COMSYMS. The size of the increase is the largest length attribute encountered for that .COMM symbol. The .COMM symbol is considered a definition of that name and its value is the offset of the contribution from the start of program section $$COMSYMS.

Appendix B

Diagnostic Messages

B.1 Diagnostic Messages from the vcc Command

This section lists the diagnostic messages that can be generated by the **vcc** command program. For each message, the description gives the message text, an explanation of the message, and suggested actions to correct the error.

error: no optimization allowed with debug

> **Warning:** You invoked the **vcc** command with both the **–O** option and the **–g** option. VAX C provides debugging, but not optimization and debugging at the same time.
>
> **User Action:** In the future, invoke the debugger without the **–O** option to avoid this error.

error: unable to execute the assembler

> **Fatal:** Either the as assembler is not available on the system or there is a protection violation that prevents its use.
>
> **User Action:** Check that the as assembler is available and that its protection is set to r-x.

error: unable to execute the compiler

> **Fatal:** Either the VAX C compiler is not available on the system or there is a protection violation that prevents its use.
>
> **User Action:** Check that the VAX C compiler is available and that its protection is set to r-x.

error: unable to execute the linker

> **Fatal:** Either the linker is not available on the system or there is a protection violation that prevents its use.
>
> **User Action:** Check that the linker is available and that its protection is set to r-x.

error: unable to execute the om

> **Fatal:** Either the om utility is not available on the system or there is a protection violation that prevents its use.
>
> **User Action:** Check that the om utility is available and that its protection is set to r-x.

error: unable to execute the preprocessor

Fatal: Either the c preprocessor (cpp) is not available on the system or there is a protection violation that prevents its use.

User Action: Check that cpp is available and that its protection is set to r-x.

error: will overwrite ****

Fatal: The listing file must not have a .c extension. Execution is halted so that the specified file is not overwritten.

User Action: Change the listing file name so that the extension is not .c.

B.2 Diagnostic Messages from the VAX C Compiler

This section lists the VAX C compiler diagnostic messages. The format of the error messages is as follows:

"file-name", line nnn: %severity-mnemonic, msg

The file-name is replaced by the name of the source file that generated the message, and nnn is replaced by the line number that identifies the location in the source file where the error was detected. The error severity code is followed by a hyphen that is followed by a brief mnemonic message abbreviation that provides a key to the alphabetized list that appears later in this section. The last part of the message, msg, is replaced by the message text that is associated with the mnemonic.

The severity can be one of the following single-letter codes representing the meanings shown:

F Fatal
E Error
W Warning
I Informational
S Success

For each message, the descriptions that follow give the mnemonic, the message text, an explanation of the message, and suggested actions to correct the error.

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** option on the **vcc** command line. You may want to do this so that the compiler broadcasts only the most severe messages to the terminal. For more information about the [NO]WARNINGS option, see Chapter 2.

ANACHRONISM, The "****" operator is an obsolete form, and may not be portable.

Informational: You used an old-style assignment operator such as =+ or =*.

User Action: For the program to be portable, reverse the order of the operator parts. For example, change =+ to += and change =* to *=. VAX C still supports the old-style operators, but they may not be supported by other C compilers, and they are not guaranteed to be supported in future releases of VAX C.

ARGINVSTRPTR, The *** argument of "***" built-in function is not a pointer to structure or union with size: 1, 2, or 4 bytes.

Error: A built-in function that takes a **struct** argument was not passed a **struct** of the appropriate size.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments.

ARGLISTOOLONG, Function reference specifies an argument list whose length exceeds the VAX architecture limit.

Error: The size of your argument list in the function call exceeded 255 longwords.

User Action: Rewrite the function definition and function call with a list whose member(s) take less space; for example, by passing floating-point and structure arguments by reference rather than by value. Recall that floating-point arguments occupy two longwords, and that structures passed by value occupy as many longwords as are necessary to contain the whole structure.

ARGNOFLOAT, The *** argument of "***" builtin function may not be floating point. The argument has been converted to an integer.

Warning: An argument to a built-in function has a floating-point type when it should have an integer type.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. If you wish to pass a **float** argument, use an explicit cast.

ARGNOTINTPTR, The *** argument of "***" builtin function is not a pointer to integer.

Error: An argument to a built-in function does not have the required type of pointer to some type of integer.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. Check the arguments for missing address-of operators (&).

ARGNOTLVALUE, The *** argument of "***" builtin function is not an lvalue.

Error: An argument that is required to be an lvalue is a non-lvalue expression.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. Make sure the appropriate arguments are lvalues.

ARGNOTPTRVAL, The *** argument of "***" builtin function is not a pointer.

Error: An argument that is required to be some type of pointer does not have a pointer type.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. Check the arguments for missing "address of" operators (&).

ARGOVERFLOW, Length of the argument list for macro "****" exceeds buffer capacity; overflowing argument(s) considered to be null.

Warning: The total length of the arguments in a macro reference exceeded the compiler's capacity to store the arguments prior to substitution.

User Action: Shorten or eliminate one or more arguments.

ARGREADONLY, The *** argument of "*** builtin function is read-only.

Error: An argument that is used by the function to modify memory is a pointer to const or read-only memory.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. Make sure that arguments that the function uses to change memory point to writeable memory.

ARGSTOOFEW, Argument list for builtin function "***" contains too few arguments; the builtin function is being ignored.

Error: Not all of the required arguments were specified.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments.

ARGSTOOMANY, Argument list for builtin function "***" contains too many arguments; excess arguments ignored.

Warning: A function was called with extra arguments.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments.

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.

BADPSECT, The program section (psect) specified by this statement has conflicting 'nowrite' attributes with another definition of the same program section.

Warning: You specified two or more references to the same program section, and the attributes of the references do not correspond.

For example, this message appears when two **globaldef** definitions exist for the same name, but only one specifies the **readonly** storage class.

User Action: Make all references to a program section consistent.

BUGCHECK, Compiler bug check during ****. Submit an SPR with a problem description.

Fatal: An internal error occurred during the specified phase of compilation.

User Action: Gather as much information as possible about the conditions under which the error occurred, including the phase of compilation, and submit an SPR (see the *Basic Installation Guide*).

BUILTARGCONV, The *** argument of "***" builtin function has been converted from pointer to arithmetic type.

Warning: An argument that should have an integer or floating-point type had a pointer type.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments. If you want to pass a pointer argument to an arithmetic argument, use an explicit cast.

CANTINLINECALL, Can't inline this call to "****" as requested because not enough actual parameters are supplied in the call.

Informational: The number of parameters supplied in a call to the function is fewer than the number of formal parameters declared and used in the function. Function calls that do not supply enough parameters will not be expanded inline.

User Action: Change the call so that all necessary parameters are supplied, or eliminate unneeded formal parameters from the function.

CANTINLINECALL, Can't inline this call to "****" as requested because an offset into a by value parameter exceeds size of actual.

Informational: The actual value of a parameter provided in a call was smaller in size than the corresponding formal parameter of the function. Use of the formal parameter requires the full amount of storage. This indicates that the type of the formal parameter does not match the type of the actual value provided in the call.

User Action: Change either the formal parameter or the actual value provided in the call so that the type of the formal parameter matches the type of the actual value.

CANTINLINEPROC, Can't inline "****" as requested because a variable offset into a by value parameter is used.

Informational: A formal parameter is referenced with a run-time variable subscript. This is usually a parameter of type **struct** containing a field that is an array. Functions that use formal parameters in this way will not be expanded inline.

User Action: Pass a pointer to the **struct** instead of the **struct** itself, or remove the pragma that requests that the function be expanded inline.

CANTINLINEPROC, Can't inline "****" as requested because it declares an exception handler.

Informational: It was requested that a function be expanded inline. However, that function declares an exception handler. Since the function would not have a call frame, it cannot have an exception handler if it is to be expanded inline.

User Action: Eliminate the exception handler, or remove the pragma that requests that the function be expanded inline.

CANTINLINEPROC, Can't inline "****" as requested because it takes the address of a passed by value parameter.

Informational: The function uses operators such as & to take the address of a formal parameter, or uses the *varargs* package. These practices prevent inline expansion of the function because it may store parameters in registers (which have no address) after inline expansion, and because you may have been relying upon the parameters being adjacent to each other in memory, which will not be true after inline expansion.

User Action: If possible, code the function without using the address of the parameter, or if an address is needed, then change the formal parameter to be a pointer to the value. If the *varargs* package is used, then remove the pragma requesting that the function be expanded inline.

CASECONSTANT, Case label value is not a constant expression.

Error: You specified a value in a **case** label that was not a constant.

User Action: Replace the **case** value with a valid constant expression.

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.

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.

CONBUILTARG, Constant expression required for "****" argument of "****" builtin function.

Error: Some built-in functions require that certain arguments be constants or expressions that the compiler can evaluate at compile time to produce a constant. If a nonconstant expression is used for any such argument, this error message is issued.

User Action: Replace the offending argument expression with a constant. If the structure of the program requires that the built-in function be called with different values that can only be calculated at run time, consider using a **switch** statement to call the built-in function with different (constant) arguments on the basis of the run-time expression.

CONFLICTDECL, This declaration of "****" conflicts with a previous declaration of the same name.

Warning: The compiler determined that both declarations see 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. 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 see 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.

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 0 was encountered in a constant expression. The expression was replaced by 0.

User Action: Make sure that no divisors in the expression can evaluate to 0.

DUPCASE, Duplicate case label value "****".

Error: You specified more than one **case** for the indicated value in a **switch** statement. (The cases must be unique.)

User Action: Change the **case** labels and combine the cases, or both, as appropriate.

DUPDEFAULT, Duplicate default label.

Error: You specified more than one default case in the same **switch** statement.

User Action: Combine the cases or make other changes necessary to eliminate the duplicate(s).

DUPDEFINITION, Duplicate definition of "****".

Warning: The named definition appeared more than once in the program.

The two definitions are essentially the same. Both definitions specify the same data types and organizations, but there may be differences in the values, initializers, or array bounds. If the name is a function, there may be a difference in the number or types of parameters, or in the contents of the function body.

User Action: The purpose of this message is to call a possible programming error to your attention.

DUPINLINEFUNC, Duplicate [no]inline function "****".

Warning: You duplicated a function name in one or more pragma declarations.

User Action: Change the name of the function declaration.

DUPLABEL, Duplicate label "****".

Error: You specified duplicates of the indicated label in the same function. (Label identifiers must be unique within a function definition.)

User Action: Rewrite the labels (and the **goto** statements that see them) to eliminate the duplicates.

DUPLISTITEM, Duplicate list item "****" ignored.

Warning: You specified the same name more than once in a list of arguments to a **#pragma** directive. For example, in the following **#pragma**, the second appearance of variable a is redundant and is ignored:

```
#pragma noinline (a, b, a)
```

Similarly, the second occurrence of variable y in the following example is redundant, as argument lists for some **#pragma** directives are concatenated:

```
#pragma noinline (x, y)
      .
      .
      .
#pragma noinline (y, z)
```

User Action: Remove the duplicate argument if it is redundant; otherwise, check for misspellings.

DUPMAINFUNC, Duplicate main function.

Warning: You defined two or more main functions in a single compilation unit.

A main function is either a function with the name main or a function with the main_program option. If the compilation unit contains more than one main function, the compiler recognizes only the first as the main function.

User Action: Make sure that there is only one main function defined in the compilation unit.

DUPMEMBER, Duplicate declaration of member "****".

Warning: You declared two members with the same name in the same structure.

User Action: Rename one of the members or remove one of the member declarations.

DUPPARAMETER, Duplicate parameter "****" ignored.

Warning: The stated function parameter occurred more than once in the function's formal parameter list. For example:

```
funct(a,b,c,a) { }
```

All occurrences of the parameter after the first are ignored.

User Action: Remove or change the duplicate parameter identifier.

ENUMCLASH, Mismatched enum type in "****" operation.

Warning: The indicated operation combined an **enum** variable or value with a value that has a nonmatching type. The compiler issues this message if you specify the **–V standard=portable** option on the **vcc** 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 you do not omit parameters 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).

EXTRATEXT, Extraneous text in preprocessor directive ignored.

Informational: Extraneous text appeared in the directive. For example:

```
#endif ABC
```

The compiler issues this message if you specify the **–V standard=portable** option on the **vcc** 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 any 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 exists. See if the include method you used for this file will search for the file in the place you expect.

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 $0.29 \times 10_{-38}$ to $1.7 \times 10_{38}$.

FUNCNOTDEF, Static function "****" is not defined in this compilation; assumed to be external.

Warning: The indicated static function declaration did not see an existing definition. The compiler treated the function as external.

User Action: Remove the storage-class specifier **static** in the function declaration or use the specifier in the appropriate function definition.

GLOBALENUM, Enumerators may not be initialized when declared with "globalref".

Warning: You tried to specify the values of enumeration constants in a declaration of an **enum** variable with the **globalref** storage-class specifier.

You must define these values elsewhere, in a **globaldef** declaration, and you must not initialize them in the **globalref** declaration.

User Action: Remove all initializing values from the **globalref** declaration.

IFEVALERROR, **** while evaluating #if or #elif expression; "true" expression assumed.

Warning: The substitute text is Stack overflow or Divide by 0.

User Action: For stack overflow, reduce the complexity of the expression. Make sure that no divisors are 0.

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.

INCBUILTARG, Incorrect type for *** argument of "***" builtin function.

Error: An argument to a built-in function has the wrong type.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments.

INLINCONF, Previous inline or noinline pragma for "****" conflicts with this pragma.

Warning: You used both an inline pragma and a noinline pragma specifying conflicting inline specifications for one particular function.

User Action: Determine whether you want the function to be expanded inline, and remove the conflicting pragma.

INSBEFORE, Inserted **** before ****.

Warning: The compiler tried to recover from a syntax error by inserting a token into the source.

User Action: Correct the syntax.

INSMATCH, Inserted **** to match **** on line ****.

INSMATCH, Inserted **** to match **** inserted earlier.

Warning: The compiler tried to recover from a syntax error by inserting a macro to match a previous macro in the source code. The previous macro may or may not have been inserted by the compiler.

User Action: Make sure that you match all parentheses, brackets, and braces.

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.

User Action: You must specify an integer constant.

INTVALREQ, Noninteger value used incorrectly in a **** ; converted to integer.

Warning: You used a noninteger value in a **switch** statement or a **case** label. The value has been converted to integer.

User Action: Specify **switch** expressions and **case** label values as integer values, or use a cast operator to make the conversion explicit.

INVAGGASSIGN, Invalid aggregate assignment.

Error: You tried to assign an array to another array or to assign structures or unions of different sizes.

User Action: Correct the assignment.

INVALIDIF, "****" is not a valid constant or operator in a #if or #elif expression; "true" expression assumed.

Warning: You used an invalid construction in an **#if** or **#elif** expression, which is assumed to be true.

User Action: Correct the expression.

INVALIGNSPEC, Invalid alignment specification ignored.

Warning: You specified an alignment option that was not in the range allowed. The compiler ignored the specified option.

User Action: Correct the alignment specification.

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.

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 0.

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 **-V**"SHOW=SYMBOLS" option on the **vcc** command line helpful to diagnose this error.

INVASSIGNTARG, Invalid target for assignment.

Error: You specified, as the left operand of an assignment operator, an expression that was not valid for assignment. For example, you may have tried to assign something to an array, to a function, to a constant, or to a variable declared with the **readonly** storage-class modifier.

User Action: Make sure that the target is appropriate for assignments.

INVBREAK, Invalid use of the "break" statement.

Error: You used **break** outside the body of a **for**, a **while**, a **do**, or a **switch** statement.

User Action: Remove the **break** statement, or check that any braces in recent loops or **switch** statements are properly balanced.

INVBUILTIN, The "***" builtin function call is being ignored; it has invalid argument(s).

Error: A call to a built-in function contains errors. This message usually follows other error messages describing errors in the argument expressions.

User Action: Correct any errors listed before this one. Make sure that the function is called with the correct number and types of arguments.

INSMATCH, Inserted **** to match **** on line ****.

INSMATCH, Inserted **** to match **** inserted earlier.

Warning: The compiler tried to recover from a syntax error by inserting a macro to match a previous macro in the source code. The previous macro may or may not have been inserted by the compiler.

User Action: Make sure that you match all parentheses, brackets, and braces.

INVCMDVAL, "****" is an invalid command qualifier value.

Fatal: The indicated CC command option value was acceptable to the command language interpreter, but it is meaningless to VAX C; for example, LIST_OPTS is an invalid value for /SHOW, but it is accepted by the **vcc** command.

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 or 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**, a **while**, or a **do** statement.

User Action: Remove the **continue** statement, or check that any braces in recent loops are properly balanced.

INVCONVERT, The source or target of a conversion is noncomputational.

Error: One of the operands in an expression could not be converted as specified. For example, you tried to cast some object to a structure.

User Action: Correct the expression or cast.

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. For example:

```
#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 specified one that was 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 main_program option.)

User Action: Check the spelling of the option or the syntax of the function definition.

INVHEXCHAR, Invalid hexadecimal character value; high-order bits truncated.

Warning: An escape character specified in hexadecimal exceeded the limit of a 1-byte character.

User Action: Correct the hexadecimal constant to represent a valid escape character.

INVHEXCON, Hexadecimal constant contains an invalid character.

Error: You specified an invalid hexadecimal constant, such as 0xG.

User Action: Correct the constant.

INVIFNAME, Missing or invalid name in #ifdef or #ifndef preprocessor directive; "true" assumed.

Warning: You specified no name or a syntactically invalid one in the directive; the result of the test is assumed to be true.

User Action: Correct the directive.

INVINAGGASN, Invalid "***" built-in function call; structure or union arguments are not of same size.

Error: A built-in function that requires two or more arguments be of the same size was called with arguments of different sizes.

User Action: Correct the call to the built-in function to pass the correct number and type of arguments.

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.

INVMAINRETVAL, Return value of main function is not an integer type.

Warning: You declared a main function with a return value that was 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.

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.

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 **int** data-type modifier is ignored.

User Action: Remove or change the data-type modifier.

INVOCTALCHAR, Invalid octal character value; high-order bits truncated.

Warning: The octal value in an escape sequence was too large, as in '\477'. Its high-order bits were truncated.

User Action: Correct the value.

INVOPERAND, Invalid **** operand of a "****" operator.

Error: You specified an invalid operand for the indicated operator.

This message is issued for arithmetic and bitwise operators if the operand is noncomputational (such as a structure). For other operators (such as the increment operator), the compiler issues the message if the operand is not an lvalue. For binary operators, the substituted text indicates which operand, left or right, is invalid.

User Action: Make sure that the operand is the proper type for the operator, and that it is an lvalue.

INVPPKEYWORD, Missing or invalid keyword in preprocessor directive; directive ignored.

Warning: You wrote a directive with no keyword. For example:

```
# ABC
```

The directive 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 tried to perform an invalid arithmetic operation on a pointer or pointers. The only valid arithmetic operations allowed with pointers are addition and subtraction.

For addition, the only forms allowed are as follows:

```
pointer  +  integer
pointer +=  integer
```

For subtraction, the only forms allowed are as follows:

```
pointer  -  integer
pointer -=  integer
pointer  -  pointer
```

In the last form, both pointers must point to objects of the same size.

User Action: Make sure that the expression conforms to one of the permitted forms previously listed. If necessary, cast one or both operands to a compatible type.

INVSUBUSE, Invalid use of subscripting.

Error: You specified a subscript in reference to a bit field.

User Action: Correct the syntax. If the structure containing the bit field is an array, you must specify the subscript(s) with the qualifier instead of the member name.

INVSUBVALUE, Invalid subscript value.

Error: You specified a subscript value that is not of an integer type.

User Action: Change or cast the value to an integer type.

INVTAGUSE, Invalid use of tag "****".

Error: You used a previously defined tag name in a declaration of a different type. For example:

```
enum    color  {red, green, blue};
struct  color  *cp;
```

You may only use a given tag with one of the **enum**, **struct**, or **union** types. 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 tried 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: You used the **void** keyword in a function prototype parameter list where it is not the only item in the list.

User Action: Either eliminate the use of **void** or eliminate the extra parameter types in the parameter list.

LISTTOOLONG, List in #pragma preprocessor directive is too long; directive ignored.

Warning: You have specified more than 128 items in the list. The entire directive was ignored by the compiler.

User Action: Split the list into separate directives.

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 the merged tokens.

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. Otherwise, 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 that appeared earlier in the source file contains a different number of parameters than this declaration.

User Action: Determine which of the declarators 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 balance the parentheses in the expression.

MISPRAGMASTAND, Mismatched #pragma standard preprocessor directive(s)

Informational: The compiler detected more occurrences of the **nostandard** pragma than it did the **standard** pragma.

User Action: Check that each **nostandard** pragma has a matching **standard** pragma, both in the main source file and in any included files.

MISSENDIF, Missing #endif preprocessor directive(s).

Error: The compiler did not encounter an **#endif** line for the most recent **#if**, **#ifdef**, or **#ifndef** preprocessor directive.

User Action: Make sure that all the 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 0.

User Action: Correct the constant.

MISSPELLED, Replaced **** with ****.

Warning: You misspelled a reserved word.

User Action: Correct the spelling.

MISWIDETYPE, The prototype for this function does not specify the default widened type for the parameter.

Error: A prototype was declared with a parameter having a type that is, by default, widened with old-style function definitions. For example, a **float** is, by default, sized to a **double** for old-style function definitions. If a prototype is in scope with a size of **float**, then the argument will not have the size that the function expects.

User Action: Correct the declaration in the prototype to specify the larger, widened type. If the type is a **float**, then specify **double**.

MODZERO, Constant expression includes mod 0; the result has been replaced with 0.

Warning: The constant expression had an invalid mod expression, such as 5 % 0. The result was 0.

User Action: Correct the expression (but note that its operands must not be floating-point operands).

NAMETOOLONG, Identifier name exceeds 255 characters; truncated to "****".

Warning: VAX C identifiers are limited to a length of 255 recognized characters.

User Action: Shorten the indicated identifier.

NESTEDCOMMENT, Nested comment encountered.

Informational: The compiler detected an opening comment delimiter (/*) within another comment. (VAX C does not support the nesting of comments; the first ending comment delimiter (*/) encountered ends the comment.)

User Action: Check that you have not misplaced a comment delimiter and accidently turned necessary code into comments.

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 a **float** or **double** type.

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: To create a valid function prototype, either eliminate the type specifiers or add type specifiers to the identifiers that are missing them.

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 issues this message if you use the **–V standard=portable** option on the **vcc** command line.

User Action: Make sure that the escape sequence contains only octal digits. If the 8 or 9 is separate from the escape sequence, but must immediately follow it, then pad the escape sequence to three digits using leading zeros.

NONOCTALESC, Escape sequence in a character or string constant starts with a nonoctal digit.

Warning: The first of three digits of an escape sequence was an 8 or 9. In this case, the backslash is ignored, and the 8 or 9 was treated as a character. The compiler issues this message if you use the **–V standard=portable** option on the **vcc** command line.

User Action: Make sure that the compiler resolves 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. The compiler issues this message if you specify the **–V standard=portable** option on the **vcc** 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. The compiler issues this message if you specify the **–V standard=portable** option on the **vcc** 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 **–V standard=portable** option on the **vcc** 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 specify the **–V standard=portable** option on the **vcc** command line.

User Action: If you do not require portability, no action is necessary.

NONPORTCVT, Conversions between pointers and integers may not be portable.

NONPORTCVT, Conversions between pointers to different types may not be portable.

Informational: You converted a pointer or an address expression to an integer type or to a different pointer type, or an integer expression or a nonzero integer constant to a pointer type. Such usage may not be portable and is not recommended. The only portable assignments are between pointers to objects of the same type or conversion of the integer constant 0 to any pointer type. This message is issued only if you specified **–V standard=portable** on the **vcc** command line.

User Action: Use an explicit cast to perform the conversion.

NONPORTINIT, Automatic initialization for "****" may not be portable.

Informational: You initialized an array or structure of the **auto** storage class. The compiler issues this message if you use the **–V standard=portable** option on the **vcc** 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 issues this message if you use the **–V standard=portable** option on the **vcc** command line.

User Action: No action is necessary if you do not require compatibility with other C compilers.

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 right arrow (–>) operator, the qualifying expression to the left of the right arrow 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 the **-V standard=portable** option on the **vcc** 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 one of the VAX C specific data types **variant_struct** or **variant_union**. The compiler issues this message if you specify the **-V standard=portable** option on the **vcc** 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 that the member belongs to.

The reference is valid, because the member name is unique and the offset can be clearly resolved. 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 is not performed and the directive containing it is ignored. If the directive is **#if** or **#elif**, the expression is 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 has no way to determine which member definition to use as an offset, a message is 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.

PARAMNOTUSED, Macro parameter "****" is not referenced in the definition.

Warning: A macro definition had more parameters than appeared in its substitution. For example:

```
#define m(a,b,c) a*b
```

User Action: Specify the extra parameter in the substitution or, if it is extra, delete it from the parameter list. (This is a possible programming error.)

PARAMREDECL, This declaration of "****" overrides a formal parameter.

Warning: Your source program contained a redeclaration of one of the function's formal parameters. For example:

```
f(a) { int a; }
```

You cannot reference the parameter from within the function.

User Action: If the declaration is misplaced, move it to a position between the function header and the left brace at the beginning of the function body. Otherwise, rename one of the identifiers.

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 trying to read a continuation of a preprocessor directive.

User Action: Check for nonterminated comments, character strings, and other constructs that can span several lines of code.

PRAGMASYNTAX, Syntax error in #pragma preprocessor directive; directive ignored.

Warning: You have incorrectly coded the directive.

User Action: Correct the error. Check for misspellings or punctuation errors.

PTRFLOATCVT, Operand of pointer addition or subtraction converted from floating-point to integer.

Warning: You combined a pointer operand with a floating-point value. For example:

```
int i,*ip;
.
.
.
i = ip + 2.;
```

User Action: Make sure that pointers are used only with other pointers or with integers; in the previous example and similar situations, remove the decimal point from the literal constant.

QUALNOTLVALUE, The qualifier for "****" is not a valid lvalue.

Error: In a reference to a structure or union member accessed by the period operator (.), the qualifying expression to the left of the period must be an lvalue.

User Action: Correct the qualifying expression.

QUALNOTSTRUCT, The qualifier for "****" is not a structure or union.

Informational: In a reference to a structure or union member, the qualifying expression to the left of the period operator (.) or right-arrow operator (–>) did not represent a structure or union.

User Action: Check for spelling errors.

REDEFPROTO, This prototype conflicts with either the function definition or with a function prototype that appears earlier in the file.

Warning: The prototype conflicts with a previous declaration of this function in number or 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 compiler issues this message if you specify the **–V standard=portable** option on the **vcc** 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 & operator.

REGADDR, Taking the address of register variable "****" is not portable and causes its storage class to be changed to auto.

Informational: You used the unary ampersand operator (&) to take the address of a register variable. VAX C changes the storage class of the variable from **register** to **auto**. This allows the address of the variable to be taken. The message is used if you specified the **–V standard=portable** option on the **vcc** command line.

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:

- **defined**
- __DATE__
- __FILE__
- __LINE__
- __TIME__

User Action: Choose a different spelling for the identifier.

SEMICOLONADDED, Semicolon added at the end of the previous source line.

Warning: A missing semicolon was added to the line before 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 by using the **-V**"WARNINGS=NOWARNINGS" option to the **vcc** command. (see Chapter 2.)

User Action: Consider the individual messages and recompile if necessary.

SYMTABOVFL, The total number of symbol table pages exceeds the implementation's limit.

Fatal: The program was too complex.

User Action: Simplify the program by reducing the number and size of variables and other names, constants, and so forth.

SYNTAXERROR, **** Found **** when expecting ****.

Error: The syntax error shown prevented the generation of an object file.

User Action: Correct the errors shown.

TBLOVRFLW, Internal table overflow, too many procedures, external symbols (psects), or the program is too complex.

Fatal: Either the source file contains too many functions or expressions, or the compiler has overflowed its virtual address space.

User Action: Reduce the size of the source file by dividing it into smaller, separate files, or change the logic of the program to reduce the number of complicated expressions.

TOOFEWMACARGS, Argument list for macro "****" contains too few arguments; missing arguments assumed to be null.

Warning: You wrote a reference to the indicated macro with fewer arguments than were specified in its definition.

User Action: Make sure that the number of arguments in the macro reference is the same as the number of parameters in the definition.

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 account for all dimensions.

TOOMANYMACARGS, Argument list for macro "****" contains too many arguments; excess arguments ignored.

Warning: You wrote a reference to the indicated macro with more arguments than were specified in its definition.

User Action: Make sure that the number of arguments in the macro reference is the same as the number of parameters in the definition.

TOOMANYMACPARM, Parameter list for macro "****" contains too many parameters; excess parameters ignored.

Warning: The number of macro parameters in a **#define** preprocessor directive exceeded the implementation limit of 64.

User Action: Rewrite the macro definition so that it uses 64 or fewer parameters.

TOOMANYSTR, String constant contains too many characters; truncated.

Warning: You wrote a character-string constant whose length exceeded the implementation's limit of 65,535 characters.

User Action: Shorten the string.

TRUNCFLOAT, Double-precision floating-point constant cannot be converted to single precision; 0.0 assumed.

Warning: You specified a double-precision constant in an expression involving a conversion to single precision, but the constant's value was too small to be represented in single precision.

User Action: Ensure that 0 is a valid value in this context; if necessary, redeclare the conversion target as **double**.

TRUNCSTRINIT, String initializer for "****" contains too many characters to fit; truncated.

Warning: If the variable was a simple one-dimensional array, the initializer was truncated (so that the length of the initializer was array-1) and the null byte was added to the end of the array. If the array is a multidimensional array or an array within a structure, the initializer was truncated to the length of the array and a null byte was not added.

User Action: Treat arrays of characters as strings allowing for the null byte at the end of the array. Consider the special case of multidimensional arrays and arrays within structures, 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 CTRL/C or **kill** command.

User Action: None.

UNDECLARED, "****" is not declared within the scope of this usage.

Error: You referred to an undeclared variable. (You must declare variables before you use them.)

User Action: Check the spelling of the identifier, or add a declaration for it, if appropriate.

UNDECLARED, "****" is not declared prior to this #pragma preprocessor directive; directive ignored.

Warning: This directive lists an identifier that has not yet been declared. The entire directive has been ignored by the compiler.

User Action: Check the spelling of the identifier or add a declaration for it, if appropriate.

UNDEFIFMAC, "****" is not a currently defined macro; constant zero assumed.

Informational: The identifier of 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. This message is only generated if **–V standard=portable** is specified on the **vcc** command line.

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 occurred in a previous **#undef**. This message is only generated if **–V standard=portable** is specified on the **vcc** command line.

User Action: Remove the **#undef**, or, if applicable, appropriately add the correct definition of the identifier.

UNDEFSTRUCT, "****" is a structure or union type that is not fully defined at this point in the compilation.

Error: You used a name in the context of a structure or union tag, but the name is either undefined or is not yet fully defined as a tag.

User Action: Check the spelling of the name, and make sure that it is fully defined as a tag before using it.

UNEXPEND, Unexpected end-of-**** encountered in #define preprocessor directive; directive ignored.

Warning: The end of the **#define** directive or the end of the source file was encountered before the definition was complete.

User Action: Check for an incomplete comment within the definition, or check for a missing continuation of the directive.

UNEXPEOF, Unexpected end-of-file encountered in a ****.

Error: The compiler encountered the end of the source file while scanning for the end of a string constant or a comment.

User Action: Make sure that string constants and comments are properly terminated.

UNEXPPDIRX, Unexpected **** preprocessor directive encountered; directive ignored.

Warning: The specified directive occurred out of place and was ignored.

User Action: Check the logic of all directives in the program to be sure that it is valid.

UNKSIZEOF, Operand of sizeof has an unknown size; 0 assumed.

Warning: The operand of a **sizeof** operator was an array whose size was unknown at compile time. A size of 0 was assumed.

User Action: Change the declaration of the array to specify the appropriate dimension bound.

UNRECCHAR, Unrecognized character ignored.

Warning: The line contained either a meaningless character or one that appears out of its proper context, such as a pound 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 the directive unrecognizable. Common errors include misspelled parameters and ambiguous abbreviations.

VARNOTMEMBER, A variant aggregate must be a member of a struct or union.

Error: You tried to specify a **variant_struct** or a **variant_union** outside 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 **-V**"SHOW=SYMBOLS" option on the **vcc** command line 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.

B.3 Diagnostic Messages from the lk Linker

Errors that occur during the linking of your program are reported by the linker. These messages may result from errors in the user program, (such as references to undefined symbols), from missing pieces needed by the linker to build the executable image, or from errors in the compiler or in the linker itself. The linker also sends informational messages in certain situations.

This section describes the messages generated by the lk linker. For information on ld linker messages, see the *ULTRIX Documentation Set.*

In order of greatest to least severity, the four classes of linker diagnostic messages are as follows:

Code	Description
F	Fatal; the linker cannot continue, so it terminates processing immediately.
E	Error; the linker cannot produce an executable image as output. However, the linker does continue processing input.
W	Warning; the cause of the error should be investigated and, if possible, corrected. The linker continues to process input and will produce an output file.
I	Informational; this message only contains information; no user action is necessary.

The following example shows how linker messages are displayed by stderr:

```
%LK-F-CONFQUAL, You have specified options on the lk command line that
are mutually exclusive.
```

Table B–1 is an alphabetical list of linker diagnostic messages. For each message, the table gives a mnemonic, the class of the message, the message text, and an explanation of the message.

Table B–1: Linker Diagnostic Messages

Mnemonic	Error	Message Text and Meaning
ARZEROSYMS	W	an object library 'pathname' member 'name' with an empty symbol table The named member of the indicated object library has a 0-length symbol table.
BADCCC	F	bad compilation completion code ('n') in module 'module-name' file 'pathname' The compilation completion code in the indicated module is invalid. Recompile the module and relink the program.

(continued on next page)

Table B–1 (Cont.): Linker Diagnostic Messages

Mnemonic	Error	Message Text and Meaning
BADPSC	F	transfer address in unknown psect ('n') in module 'module-name' file 'pathname' The module specified a nonexistent psect for the transfer address. Correct the source code, recompile the module, and relink the program.
COMMDEF	I	'pathname': Definition of common 'name' size 'n' The linker encountered a definition of the named common block in the named file. This message occurs if you specify the **–y** option for that symbol on the **lk** command line.
CONFQUAL	F	conflicting qualifiers You have specified options on the **lk** command line that are mutually exclusive.
CRFERR	W	error encountered in Cross Reference The linker encountered an error while cross-referencing a symbol name. Another message will explain exactly what error was encountered.
DSTINTERR	W	internal coding error "routine_name", symbol 'name' The linker encountered a problem while processing the indicated debugger symbol. Submit an SPR on the linker.
EMPTYFILE	W	no modules found in file 'pathname' The named file contains no object code.
EOMFTL	F	link abort specified in module 'module-name' file 'pathname' The end-of-module record in the indicated module specifies ending the link. Correct the errors in the module and relink the program.
EOMSTK	W	'n' UL items left on Linker internal stack in module 'module-name' file 'file-spec' This message indicates an internal error either in the compiler or in the linker. Submit an SPR either on the compiler that generated the module or on the linker.
ERRORISUE	E	completed with errors The link completed, but errors were detected. Correct the errors and relink the program.
ERRORS	E	compilation errors in module 'module-name' file 'pathname' The indicated module generated errors during compilation. The linker will continue, but it is unlikely that the resulting program will run correctly. Correct the source code errors, recompile, and relink the program.
EXCPSC	F	more than 65535 psects defined in module 'module-name' file 'pathname' The indicated module defines too many psects. Correct the problem and relink the program.

(continued on next page)

Table B–1 (Cont.): Linker Diagnostic Messages

Mnemonic	Error	Message Text and Meaning
EXTDEF	I	'pathname': Definition of external 'entity' 'entity-name' The linker encountered a definition of the named entity in the named file. This message occurs if you specify the **-y** option for that entity on the **lk** command line.
EXTREF	I	'pathname': Reference to external undefined 'name' The linker has just encountered a reference to the indicated name in the named file. This message occurs if you specify the **-y** option for that name on the **lk** command line.
FAOFAIL	W	FAO system service failure The linker encountered an error while calling the Formatted ASCII Output (FAO) system service. Further messages will tell exactly what error was encountered.
FATALERROR	F	fatal error message issued The link completed but a fatal error message was issued.
FILETRACE	I	now processing file: 'pathname' The linker is now processing the indicated file. This message occurs if you specify the **t** option on the **lk** command line.
FORMAT	F	file has illegal format One of the files on the command line is neither an object module nor an object library. Correct the file references and reissue the **fort** command.
GSDTYP	W	illegal GSD record type 'type' in module 'module-name' file 'pathname' The indicated module in the indicated file is corrupt; it contains an illegal Global Symbol Dictionary (GSD) record. Correct the problem and relink the program.
ILLARFOR	F	object library 'pathname' has illegal format The linker detected a format error in the indicated object library. Rebuild the library file.
ILLDEFOFF	F	has a symbol definition offset greater than corresponding segment size The file or member defines a symbol whose offset is outside the program segment in which the symbol is defined. Submit an SPR on the compiler that generated the file or member.
ILLFMLCNT	W	minimum argument count 'n' exceeds maximum ('n') in formal specification of symbol 'name' in module 'module-name' file 'pathname' The object records that describe the indicated symbol specify inconsistent values for the minimum and maximum permissible argument counts for calling the routine. Submit an SPR on the compiler that generated the module.

(continued on next page)

Table B–1 (Cont.): Linker Diagnostic Messages

Mnemonic	Error	Message Text and Meaning
ILLMEMMAG	F	object library 'pathname' has a member with an illegal ar_hdr magic number A member in the indicated object library has a bad magic number field. Rebuild the library file.
ILLNAMELEN	F	'entity' 'name' name length ('n') is illegal—not 1 to 'n' in module 'module-name' file 'pathname' The length of a psect, module, or symbol name is not in the specified range. Correct the length and relink the program.
ILLOBJFOR	F	has illegal format The indicated object file or member is not formatted properly. Rebuild the object file and relink the program.
ILLOBJMAG	F	has an illegal magic number The indicated object file or member has a magic number field that contains a value other than OMAGIC, NMAGIC, or ZMAGIC. Rebuild the object file and relink the program.
ILLODDSEG	F	has an odd length text or data segment The text and data segments of an object file or member must have even lengths. Either the file or member is corrupt or there is a bug in the compiler that generated the file or member. First, correct the file or member and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the file or member.
ILLRANTOC	F	ranlib file 'pathname' has an illegal table of contents The indicated library's table of contents is corrupt. Run ranlib on the library and relink the program.
ILLRECLEN	W	illegal record length ('n') in module 'module-name' file 'pathname' The indicated module contains a record that is too long or that is inconsistent with the record type. This can be caused by a corrupt file or by a compiler bug. First, correct the file and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the file.
ILLRECTYP	W	illegal record type ('n') in module 'module-name' record 'n' file 'pathname' The indicated object record contains a bad type field. This can be caused by a corrupt file or by a compiler bug. First, correct the file and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the file.

(continued on next page)

Table B–1 (Cont.): Linker Diagnostic Messages

Mnemonic	Error	Message Text and Meaning
ILLRELOFF	F	has a relocated field outside the corresponding segment The relocated value for a field places it outside the segment in which it is defined (for example, a text segment symbol whose relocated value places it in the data segment). Correct the source code and relink the program.
ILLRELSYM	F	has a relocation field that points to an illegal symbol A relocation field in the file or member points to a symbol that is not a data definition symbol. Submit an SPR on the compiler that generated the file or member.
ILLSTATEVAL	F	current state out of range in lnk$nxtultrec The linker encountered an internal error. Submit an SPR on the linker.
ILLSTRIDX	F	has an illegal string index in a stab entry The linker encountered a symbol table entry with an illegal string table index. Submit an SPR on the compiler that generated the file or member.
ILLSYMNAM	F	file 'pathname' contains a symbol that exceeds the maximum size The indicated file contains a symbol name longer than 255 characters. Reduce the size of the symbol and then recompile and relink the program.
ILLTIR	W	illegal relocation command ('n') in module 'module-name' record 'n' file 'pathname' The indicated object record contains a bad relocation command. This can be caused by a corrupt file or by a compiler bug. First, correct the file and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the file.
ILLVPS	W	illegal position ('n') or size ('n') in STO_VPS command in module 'module-name' file 'pathname' The indicated object record contains a bad STO_VPS command. This can be caused by a corrupt file or by a compiler bug. First, correct the file and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the file.
ILLZMAGSEG	F	has an illegal zmagic text or data segment size (not multiple of 1024) The object file or member has a header specifying ZMAGIC format, but the text or data segment size is not a multiple of the page size (1024 bytes). Either the file or member is corrupt or there is a bug in the compiler that generated the file or member. First, correct the file or member and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the file or member.

(continued on next page)

Table B–1 (Cont.): Linker Diagnostic Messages

Mnemonic	Error	Message Text and Meaning
INSVIRMEM	E	insufficient virtual memory for 'n' pages for cluster 'name' The linker was unable to allocate enough virtual memory in the resulting program to contain the entire program. Consult the link map (obtained by entering the **lk –M** command) to determine why the linker ran out of virtual memory. Then, correct the problem and relink the program.
INTERNERROR	F	internal linker error 'n', please submit spr The linker encountered an internal error. Submit an SPR on the linker, indicating the error number reported in this message.
INTSTKOV	W	linker internal stack of 'n' overflowed in module 'module-name' file 'pathname' The linker overflowed its internal stack while processing the indicated module. Submit an SPR on the linker.
INTSTKUN	W	linker internal stack of 'n' underflowed in module 'module-name' file 'pathname' The linker underflowed its internal stack while processing the indicated module. Submit an SPR on the linker.
INVDSTREC	W	invalid VAX DEBUG Symbol Table record, type 'n' subtype 'n' The linker encountered the indicated invalid debugger record. Either the object file is corrupt or there is a bug in the compiler that generated the object file. First, correct the file and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the file.
LIBNAMLNG	W	library module name 'name' has illegal length ('n') The indicated module name is too long. Shorten the name and then recompile and relink the program.
LIBNOTFND	F	'pathname' not found in /lib, /usr/lib, or /usr/local/lib The indicated library, specified using the **fort –l** command, was not found in any of the expected directories. Either move the file or specify a pathname in the **fort –l** command.
LOCDEF	I	'pathname': definition of 'entity' 'entity-name' The linker encountered a local definition of the named entity in the named file. This message occurs if you specify the **–y** option for that entity on the **lk** command line.
LOCREF	I	'pathname': Reference to undefined 'name' The linker has just encountered a local reference to an undefined symbol in the named file. This message occurs if you specify the **–y** option for that symbol on the **lk** command line.

(continued on next page)

Table B–1 (Cont.): Linker Diagnostic Messages

Mnemonic	Error	Message Text and Meaning
MEMBUG	F	memory (de)allocation bug 'n', %X'n', 'n' The linker encountered an internal error allocating or deallocating dynamic memory. Submit an SPR on the linker.
MEMFUL	E	insufficient virtual address space to complete this link There was insufficient process virtual address space or swap file space to complete the link. Either increase your virtual address space or swap file space, or decrease the size of the program you are trying to link.
MODNAM	F	module 'name' name length is illegal—not 1 to 'n' The length of a module name is not in the specified range. Correct the name length and relink the program.
MULDEF	W	symbol 'name' multiply defined in module 'module-name' file 'pathname' The linker encountered a definition for the named symbol in the indicated module, but the symbol is defined. Correct the source code, so the symbol is defined only once, then relink the program.
MULDEFPSC	W	psect 'name' multiply defined in module 'module-name' file 'pathname' The indicated module defines the named program section more than once. Submit an SPR on the compiler that generated the module.
MULPSC	W	conflicting attributes for psect 'name' in module 'module-name' file 'pathname' The named program section is defined with different attributes in different modules of your program. Correct the source code so that all instances of the same psect have the same attributes, then relink the program.
MULTFR	W	multiply defined transfer address in module 'module-name' file 'pathname' More than one object module specifies a transfer address for the program. Delete all but one transfer address and relink the program.
NOEOM	W	no end of module record found in module 'module-name' file 'pathname' The indicated module does not contain an end-of-module record. This is due to a corrupt file or a bug in the compiler that generated the file. First, correct the file and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the file.
NOEPM	W	undefined entry mask of symbol 'name' referenced in module 'module-name' file 'pathname' A .MASK directive in a VAX MACRO object module referenced an undefined symbol. Either define the symbol or delete the reference to it.

(continued on next page)

Table B–1 (Cont.): Linker Diagnostic Messages

Mnemonic	Error	Message Text and Meaning
NOIMGFIL	E	image file not created The linker did not create an output program file. Other error messages explain why it did not create the file.
NOMODS	F	no input modules specified (or found) The linker did not encounter any object modules in any of the files specified on the command line. Correct the command line and reenter the command.
NOPSCTS	F	no psects defined in module 'module-name' file 'pathname' The indicated module does not contain any program sections.
NOTOBJLIB	F	file 'pathname' is not an object library The indicated file is not an object library, but the linker expected it to be one.
NOTpsect	W	relocation base set to other than psect base in module 'module-name' file 'pathname' A "set relocation base" command in the indicated module specified a relocation base other than the base of the psect. Submit an SPR on the compiler that produced the module.
NOTXTENT	I	Entry specified is not defined in text. The entry point name specified in the **fort –e** command is not in the program text section of your program. Instead, it is in the data or uninitialized data section. The program entry point must be in the program text section. Correct the source code and relink the program.
NUDFENVS	W	'n' undefined environment(s): This message reports the undefined environments encountered during the link.
NUDFLSYMS	W	'n' undefined module-local symbol(s): This message reports the undefined module-local symbols encountered during the link.
NUDFSYMS	W	'n' undefined symbol(s): This message reports the undefined global symbols encountered during the link.
OLDTOC	W	ranlib library file 'pathname' table of contents not updated since last modification, re-run ranlib on it The timestamp on the library's table of contents is older than the modification date of the library itself. Run ranlib on the file to update the table of contents.
OVRALI	W	conflicting alignment on overlayed psect 'name' in module 'module-name' file 'pathname' The named psect was defined in multiple modules with different alignments. Correct the psect declarations so that they all specify the same alignment, then relink the program.

(continued on next page)

Table B–1 (Cont.): Linker Diagnostic Messages

Mnemonic	Error	Message Text and Meaning
PSCALI	W	psect 'name' alignment ('n') illegal in module 'module-name' file 'pathname' The module specified an illegal psect alignment. Submit an SPR on the compiler that produced the module.
PSCNXR	W	transfer address is not in executable, relocatable psect in module 'module-name' file 'pathname' The transfer address for a module must be in an executable, relocatable psect. Move the transfer address and relink the program.
RECLNG	W	file 'pathname' has a record of illegal length ('n') The file contains a record of either length 0 or longer than 2048 bytes. Either the file is corrupt or there is a bug in the compiler that generated the file. First, correct the file and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the file.
RECTYP	W	file 'pathname' record 'n' is illegal ('n') The file has a record with an illegal type field. Either the file is corrupt or there is a bug in the compiler that generated the file. First, correct the file and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the file.
RLZEROTOC	W	ranlib object library 'pathname' has an empty table of contents There are no entries in the object library's table of contents. If there are object files in the library, run ranlib to build the table of contents, then relink the program.
SEQNCE	W	illegal record sequence in module 'module-name' file 'pathname' The indicated module contains an illegal sequence of object records. Submit an SPR on the compiler that generated the file.
STALITUDF	W	Stack of undefined literal 'n' in record 'n' in module 'module-name' file 'file-name' The indicated module contains an object command to push a literal value onto the linker's internal stack, but that literal is not defined. Submit an SPR on the compiler that generated the file.
STRLVL	F	illegal object language structure level ('n') should be 'n' in module 'module-name' file 'pathname' A module header record in the indicated module specifies an invalid object language format. Either the object file is corrupt or there is a bug in the compiler that created the module. First, correct the module and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the module.

(continued on next page)

Table B–1 (Cont.): Linker Diagnostic Messages

Mnemonic	Error	Message Text and Meaning
TIRLNG	W	object command data overflows record by 'n' bytes in module 'module-name' record 'n' file 'pathname' There is a bad length field in a Text Information/ Relocation record in the indicated module. Either the object file is corrupt or there is a bug in the compiler that generated the file. First, correct the file and rebuild the program. If the error continues to occur, submit an SPR on the compiler that generated the file.
TIRNYI	W	unimplemented TIR command ('n') encountered in module 'module-name' record 'n' file 'pathname' The indicated module contains a Text Information/ Relocation record that is not yet implemented by the linker. Submit an SPR on the compiler that generated the file.
TRUNC	W	truncation error in psect 'name' offset %X'n' in module 'module-name' file 'pathname' The indicated relocatable reference specified a 1-byte or 1-word relative addressing mode, but the defined address of the symbol is longer than 1 byte or 1 word. Correct the reference by using longword relative addressing.
TRUNCDAT	W	computed value is %X'value1' value written is %X'value2' at location %X'addr' This message accompanies the TRUNC message to give more detailed information about the truncation error. 'value1' is the value that the linker tried to store. 'value2' is the truncated version that the linker was able to store. 'addr' is the virtual address in the executable program where the value is stored.
UDEFPSC	W	attempt to reference undefined psect number 'n' in module 'module-name' file 'pathname' The indicated module references a psect index that it did not define. Submit an SPR on the compiler that generated the object file.
ULTOBJFIL	F	object file 'pathname' There is a problem in the object file named by "pathname." Another message describing the problem will follow.
ULTOBJMEM	F	object library file 'pathname' member 'name' There is a problem in a member of an object library. Another message describing the problem will follow.

(continued on next page)

Table B–1 (Cont.): Linker Diagnostic Messages

Mnemonic	Error	Message Text and Meaning
USEUDFENV	W	undefined environment 'name' referenced OR undefined environment number 'n' referenced in psect 'name' offset %X'n' in module 'module-name' file 'pathname' The linker encountered a reference to an environment that was not defined in any of the object modules. Either define the environment in one of the object modules or remove the reference.
USEUDFLSY	W	undefined module-local symbol 'name' referenced in psect 'name' offset %X'n' in module 'module-name' file 'pathname' The indicated location references a module-local symbol that was not defined. Correct the source code so that either the symbol is defined or the reference is removed.
USEUNDEF	W	undefined symbol 'name' referenced in psect 'name' offset %X'n' in module 'module-name' file 'pathname' The indicated location references an external symbol that was not defined. Correct the source code so that either the symbol is defined or the reference is removed.
WARNISUE	W	completed with warnings The linker finished processing the program, but warning messages were issued.
WRNERS	W	compilation warnings in module 'module-name' file 'pathname' The compilation that produced the indicated module generated warning messages. Depending on the nature of the warnings, you may have to correct the errors that caused the warnings and recompile the module.

Appendix C

Transporting VAX C Programs Between VMS and ULTRIX Systems

This appendix describes how to transfer VAX C programs and data between VMS and ULTRIX systems. It also describes many of the differences and incompatibilities that you may need to resolve before a file—developed on one type of system (ULTRIX or VMS) and transported to a different type of system (ULTRIX or VMS)—can be used on the receiving system.

This appendix addresses the following topics:

- Section C.1 describes several methods that you can use to transport VAX C program files between VMS and ULTRIX systems.
- Section C.2 describes differences that result from the compiling and linking process on the two types of systems.

C.1 Transporting VAX C Programs and Other ASCII Files

The following sections describe several mechanisms that you can use to transfer program files between VMS and ULTRIX systems.

C.1.1 Using DECnet-ULTRIX to Copy ASCII Programs

To copy files from a VMS system to an ULTRIX system, you can use the DECnet-ULTRIX[1] copy command, **dcp**, to pull the files over the network. To do this, enter a **dcp** command with the following form:

```
% dcp vms_node/account/password::'dev:[dir]in_file' out_file
```

The definitions of the variables (in italics) in the preceding command line are as follows:

vms_node	VMS node name
account	VMS account that you will log in to
password	Password of the specified VMS account
dev:[dir]	Device (dev) and directory (dir) where the file resides
in_file	Input file name
out_file	Output file name

[1] DECnet-ULTRIX is a layered product available under a separate license for ULTRIX systems.

To copy files from an ULTRIX node to a VMS node, use the **dcp** command to push the files over the network. To do this, enter a **dcp** command with the following form:

```
% dcp in_file vms_node/account/password::'dev:[dir]out_file'
```

The variables in this command line are the same as those in the previous command line.

C.1.2 Using DECnet–VAX to Copy ASCII Programs

You can use the DECnet–VAX[2] COPY command to copy ASCII files between a VMS node and an ULTRIX node. To pull a VAX C source file from an ULTRIX node to a VMS node, enter the COPY command with the following form:

```
$ COPY ult_node"account password"::"/dev/dir/in_file" out_fil e
```

The definitions of the variables (in italics) in the preceding command line are as follows:

ult_node	ULTRIX node name
account	ULTRIX account that you will log in to
password	Password of the specified ULTRIX account
/dev/dir/	Device (dev) and directory (dir) where the file resides
in_file	Input file name
out_file	Output file name

The resulting VMS file is not created with record format VARIABLE. This causes the VMS editor (EDT) to issue the following message:

```
Input file does not have standard text file format
```

Otherwise, the file can be used as an ASCII file.

C.1.3 DEC/Shell on a VMS System — The tar Utility

You can use the DEC/Shell[3] tar utility to move ASCII files from a VMS system to an ULTRIX system, or to restore ASCII files written by tar on an ULTRIX system. When restoring files from an ULTRIX system to a VMS system, you must physically mount the tape on the tape drive from which you want to perform the restore procedure. You do not need to enter the MOUNT command because the tar utility does this automatically.

NOTE

> The tar utility cannot be used alone to transfer binary data files between a VMS system and an ULTRIX system. The data formats on the two systems are incompatible and require additional processing beyond that provided by tar.

In the following example, the function letter **c** directs the tar utility to create a new tape. Writing starts at the beginning of the tape instead of after the last file. The tar utility writes the named file(s) to the tape.

```
% tar -c foo.c
```

[2] DECnet–VAX is a layered product available under a separate license for VMS and MicroVMS systems.

[3] DEC/Shell is a layered product available for VMS and MicroVMS systems under a separate license.

The following command extracts all the files in the ./cprogs directory. By default, the device mta0: is used.

```
% tar -x ./cprogs
```

The function letter **x** directs tar to extract files from tape. Because a directory name is given as a parameter, tar recursively extracts files from the directory. If you do not specify a parameter, tar extracts the entire contents of the tape.

C.2 Compiling and Linking Considerations

The following sections describe the I/O files associated with the **vcc** (on ULTRIX systems) and **CC** (on VMS systems) compilation commands, the search paths associated with the **vcc** command, and the differences in psect usage and image sizes on the two types of systems.

C.2.1 Input and Output Files

On a VMS system, the process of compiling and linking a VAX C program generally requires two steps. For example, if you have a VAX C source program named FOO.C, the following sequence of commands results in an executable image, FOO.EXE:

```
$ CC FOO
$ LINK FOO
```

The first command invokes the VAX C compiler, which compiles the input file FOO.C, and produces the output file FOO.OBJ. The second command invokes the linker, which links the input file FOO.OBJ (along with appropriate support routines), and produces the output file FOO.EXE.

On an ULTRIX system, you can accomplish the same process by using a single **vcc** command line. The **vcc** command causes the VAX C compiler and the linker to execute with the appropriate arguments, based on its interpretation of its own arguments. Other processors (such as the ULTRIX assembler and the ULTRIX C preprocessor, cpp) can also be invoked by the **vcc** command, depending on the arguments supplied on the **vcc**command line.

The following **vcc** command is similar to the VMS command sequence previously described:

```
% vcc -o foo foo.c
```

In this example, the presence of foo.c causes the **vcc** command to invoke the VAX C compiler with the file foo.c as an input file; the compiler produces the file foo.o as output. The .c must be supplied explicitly. The **vcc** command then invokes the linker with –o foo and foo.o (and appropriate libraries) as arguments, and the linker produces the executable image foo. The **vcc** command program then deletes the intermediate file foo.o.

The **vcc** command assumes the existence of intermediate files not specified by the command line, which are created by one processor and passed to another (for example, foo.o, created by the VAX C compiler and linked by the linker). The **vcc** command derives the names of these intermediate files from its arguments. For example, the **vcc** command assumes that any file whose name ends in .c or .h is a VAX C source file that should be compiled by the VAX C compiler. The **vcc** command further assumes that the VAX C compiler will create a file of the same name (minus the .c or .h), with the file extension .o. For example, compiling foo.c produces foo.o.

C.2.2 Search Paths Used by the vcc Command

The **vcc** command looks for the processor it expects to execute, as well as support routine libraries and objects, in a sequence of directories in the ULTRIX file system. This sequence, or search path, is as follows:

- The **vcc** command looks for the VAX C compiler by trying to execute in order:
  1. The directory specified in the **–B** option, if **–t0** is specified.
  2. /usr/lib/vcc
  3. /lib/vcc
- The **vcc** command looks for the linker in the following places:
  1. The directory specified in the **–B** option, if **–tl** is specified.
  2. /usr/bin
  3. /bin
- If the **–Em** option is specified, the **vcc** command looks for the ULTRIX C preprocessor cpp in the following directories:
  1. The directory specified in the **–B** option, if **–tp** is specified.
  2. /usr/lib
  3. /lib
- The **vcc** command assumes that the C libraries are in the /usr/lib directory.

If you need to print a diagnostic message while executing either the VAX C compiler, the linker, or a user program, the message routines look for the following messages in the file:

- $fortmsgfile — if the environment variable fortmsgfile is defined
- /usr/lib/fortmsgfile — otherwise

C.2.3 Psect Differences

There are numerous differences between the way that psects are set up by VAX C on a VMS system and the way that they are set up by VAX C on an ULTRIX system.

C.2.4 Image Size Differences

The size of the executable VAX C programs is much larger on ULTRIX systems than with either VAX C programs on VMS systems or pcc programs on ULTRIX systems. There are several reasons for this as follows:

- The VMS operating system supports shareable images, in which libraries of subroutines can be shared by more than one program. This effectively reduces the size of a program's executable image for those programs that use routines contained within the shared libraries. For VAX C programs, these shareable libraries include the math library routines, Record Management Services (RMS) routines (on VMS systems), and system service routines.

Unlike the VMS operating system, the ULTRIX operating system has no comparable facility for sharing common routines; each program that uses shared libraries or system service routines must have the routines physically present as part of the program's executable image. As a result, these images are much larger on an ULTRIX system than their counterparts would be on a VMS system.

- The ULTRIX system performs demand zero compression only on the bss section of a program. (The bss section appears at the end of a program image file.) When image activation occurs on an ULTRIX system, this section is allocated zero initialized memory.

  Demand-zero compression is the extraction of contiguous, uninitialized, writeable pages from an image section and the placing of these pages into a newly created demand-zero image section.

  A demand-zero image section contains uninitialized, writeable pages that do not occupy space in the image file on disk, but which, when accessed during program execution, are allocated memory and initialized with binary zeros by the operating system.

  In pcc, the compiler does the work of separating initialized data from uninitialized data. However, many of the VAX C optimizations depend on a storage layout that prevents such separation. For this reason, no local storage for variables or arrays is allocated to bss.

C.3 Transferring Data Files Between VMS and ULTRIX Systems

You can make data file transfers between VMS and ULTRIX systems by using the DECnet dcp utility described in Section C.1.1 or magnetic tape, using the tape archive utility, tar, which is described in Section C.1.3.

Appendix D

Language Summary

This appendix describes the **vcc** command and the VAX C/ULTRIX language features.

D.1 The vcc Command

The **vcc** command is an ULTRIX command that 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.

The **vcc** command has the following form:

vcc [-*options* [*args*]]... *filename*[.*type*] [...*filename*[.*type*]] [-*options* [*args*]]

vcc Command Options:

Option	Description
–B *string*	Finds a substitute compiler, preprocessor, an assembler, and a linker in the files named by string. If string is empty, use a standard backup version.
–b	Does not pass the library file **–lc** to the linker. This is a linker option.
–c	Generates an object file with a .o file extension. The linked, executable module is not generated.
–D *name=def*	Assigns the specified value (*def*) to *name*. The preprocessor interprets this option. If a definition value is not specified, the name is set equal to 1.
–E	Runs the vcc preprocessor. The code is preprocessed, and all preprocessor directives, such as include file statements, are resolved.
–Em	Runs the cpp preprocessor and produces the makefile dependencies.
–f	Enables single-precision, floating-point arithmetic. Double-precision, floating point is the default selection. Procedure arguments are still promoted to double-precision, floating-point format.
–g	Generates additional symbol table information for the dbx debugger.
–I*dir*	Specifies the name of the directory containing the relevant include files. A search for included files whose names do not include a directory specification, occurs in: the directory of the file, the directory named by the **–I** option, and finally in directories contained in a standard list.
–K	Generates a full MAP table. This is a linker option. It may be specified on the **vcc** command line or the linker command line.

Option	Description
-l*x*	Specifies a library to include in the link process. The variable *x* is an abbreviation for the library and path name /lib/libx.a in which *x* is a string. If the library is not found, the linker searches for /usr/local/lib /libx.a. A library search starts when the library name is encountered. As a result, the placement of the **-l** within the **vcc** or linker command line is significant.
-Md	Specifies the double-precision, floating-point type as D_floating. This is the default selection. The linker also receives the **-lc** flag.
-Mg	Specifies the double-precision, floating-point type as G_floating. The linker also receives the **-lcg** flag. If you want to use the math library, with code generated with the **-Mg** option, you must link in the G_ FLOAT version of the library by specifying **-lmg** on the linker or **vcc** command line.
-o	Accepts the specified name as the final output file name. This is a linker option. It may be specified on the **vcc** or linker command line.
-O	Invokes the object code improver. The default selection is to perform object code optimization.
-p,-pg	Prepares object files for profiling. The **-pg** option also invokes a run-time recording mechanism that produces a gmon.out file. This file contains more extensive statistics.
-t [0pal]	Finds only the designated compiler, preprocessor, assembler, and linker in the files whose names are constructed by a **-B** option. In the absence of a **-B** option, these are found in the standard places.
-U*name*	Makes the specified variable undefined within the program. This option is interpreted by the preprocessor.
-v *filename.lis*	Produces the listing file, complete with a cross-reference table and machine code listing.
-V *option*	Compiles the source code using vendor-specific options.
-w	Suppresses compiler warning messages. Error messages are displayed, but warning messages are not.
-Y [option]	Compiles a file for one of the following options: SYSTEM_FIVE, BSD, or POSIX. If a parameter other than SYSTEM_FIVE, BSD, or POSIX is specified, a warning is printed and the **-Y** option is ignored. If no parameter is specified, **-Y** defaults to **-YSYSTEM_FIVE**. If multiple **-Y** options are specified, only the last option takes effect, and no warning message is generated.

D.2 Data-Type Keywords

Type Specifiers:

32-bit signed or unsigned:

int
long
long int
unsigned int
unsigned long
unsigned long int

16-bit signed or unsigned:

short
short int
unsigned short
unsigned short int

8-bit signed or unsigned:

char
unsigned char

F_floating format:

float

D_floating or G_floating format:

double
long float

Aggregate types:

struct
union
variant_struct
variant_union

Enumerated type:

enum

Type of function return value:

void

Type declaration:

typedef

Storage-class specifiers:

auto
register
static
extern
globaldef
globalref
globalvalue

Data-type modifiers:

const
volatile

Storage-class modifiers:

readonly
noshare
_align

D.3 Precedence of Operators

In the following table, the operators are listed from highest precedence to lowest. In the binary operator category, operators appearing on higher lines within the category have a higher precedence than the other binary operators.

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.4 Statements

Syntax:

[*expression*] ;

identifier : *statement*

{ [*declaration-list*] [*statement-list*] }

case [*constant-expression* | default] : *statement-list*

if (*expression*) *statement* [**else** *statement*]

while (*expression*) *statement*

do *statement* **while** (*expression*)

for ([*expression*] ; [*expression*] ; [*expression*]) *statement*

switch (*expression*) *statement*

break ;

continue ;

return [*expression*] ;

goto *identifier* ;

D.5 Conversion Rules

Arithmetic Conversion

Any operand of type:	Is converted to:

char	**int**
short	**int**
unsigned char	**unsigned int**
unsigned short	**unsigned int**
float	**double**

If operand type is:	The result and the other operands are:
double	**double**
unsigned	**unsigned**

Otherwise, both operands are:	And the result is:
int	**int**

Function Argument Conversion without Prototypes

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.6 VAX C Escape Sequences

The following table lists the 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

Use the form "\ddd" to specify any byte value (usually an ASCII code), where the digits ddd are one to three octal digits. The octal digits are limited to 0 to 7.

D.7 Preprocessor Directives

Syntax:

#define *identifier*[([*param1, . . . param2*])] *token-string*
#undef *identifier*
#elif *constant-expression*
#include *<file-path>*

#include "*file-path*"
#if *constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#[**line**] *constant string*
#[**line**] *constant identifier*
#pragma [no]builtins
#pragma [no]inline
#pragma [no]member_alignment
#pragma [no]standard

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Guide to VAX C for ULTRIX
AA-ME83B-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	□	□	□	□
Completeness (enough information)	□	□	□	□
Clarity (easy to understand)	□	□	□	□
Organization (structure of subject matter)	□	□	□	□
Figures (useful)	□	□	□	□
Examples (useful)	□	□	□	□
Index (ability to find topic)	□	□	□	□
Page layout (easy to find information)	□	□	□	□

What would you like to see more/less of? ______________________________

____

____

What do you like best about this manual? ______________________________

____

____

What do you like least about this manual? ______________________________

____

____

Please list errors you have found in this manual:

Page Description

______ ____

______ ____

______ ____

______ ____

______ ____

Additional comments or suggestions to improve this manual:

____

____

____

____

What version of the software described by this manual are you using? ____________

Name/Title ____ Dept. __________

Company ____ Date __________

Mailing Address ____

________________________ Email ____________________ Phone __________

--- Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

--- Do Not Tear – Fold Here

Cut
Along
Dotted
Line