Compaq C

Migration Guide for OpenVMS VAX Systems

Order Number: AA–Q5AVB–TE

November 1999

This guide contains information to assist *VAX C* application developers in migrating from a *VAX C* environment to a *Compaq C* environment on *OpenVMS VAX* systems.

Revision/Update Information: This revised guide supersedes the *DEC C Migration Guide for OpenVMS VAX Systems* (Order No. AA–Q5AVA–TE)

Software Version: *Compaq C* Version 6.2 or higher for *OpenVMS VAX* Systems

Compaq Computer Corporation
Houston, Texas

First Printing, May 1994
Revised, November 1999

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.

© Digital Equipment Corporation 1994 - 1999. All Rights Reserved.

Compaq, the Compaq logo, Alpha, DEC, DECnet, DECtalk, DECthreads, DIGITAL, OpenVMS, RMS, Tru64 UNIX, VAX, VAX C, CDD/Repository, VMS, and the DIGITAL logo are registered in the U.S. Patent and Trademark Office.

The following are third-party trademarks:

Motif and OSF are registered trademarks of the Open Software Foundation, Inc.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Ltd.

ZK6396

This document is available on CD-ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Preface v

1 Migrating to the Compaq C Run-Time Library

1.1 Why Migrate to the Compaq C RTL 1–1
1.2 How to Migrate to the Compaq C RTL 1–2
1.2.1 Migrating Using the Compaq C Compiler 1–2
1.2.1.1 Linking with the Compaq C RTL Shareable Images 1–3
1.2.1.2 Linking with the Compaq C RTL Object Libraries 1–3
1.2.2 Migrating VAX C Code Using Shareable Images 1–4
1.2.3 Migrating VAX C Code Using Object Libraries 1–4
1.3 Potential Migration Concerns and Solutions 1–5
1.3.1 Linking /NOSYSSHR with the Compaq C RTL Object Libraries 1–6
1.3.2 Behavior Differences Between the VAX C RTL and Compaq C RTL 1–6
1.3.3 Compaq C RTL Obsolete Features 1–9
1.3.3.1 VAXC$MALLOC_OPT and Related Routines 1–9
1.3.3.2 sys_nerr and sys_nerrlist Global Symbols 1–9
1.3.4 Debugging and the Compaq C RTL Object Library 1–9
1.4 C RTL Interoperability Concerns 1–10
1.4.1 Problems with Images that Export C RTL Context 1–12
1.4.1.1 Becoming Encapsulated—Solutions to the General Problem 1–15
1.4.1.2 Issues for File I/O 1–16
1.4.1.3 Considerations for errno 1–16
1.4.1.4 Child Process Context 1–17
1.4.1.5 Random Number Seed 1–18
1.4.1.6 Signal Context 1–18
1.4.1.7 VAXC$CRTL_INIT and Context Initialization for main Program 1–18

1.4.2 Resolving Link-Time Conflicts with Multiple C RTLs 1–19
1.4.2.1 Using VAXC$LCL.OPT 1–20
1.4.2.2 Using VAXC$EMPTY.EXE 1–21
1.4.2.3 Using DECC$EMPTY.EXE 1–22
1.4.3 Distributing Programs to Other Systems—Version Dependency 1–23
1.4.3.1 Using the Compaq C/C++ Run-Time Components Kit . . . 1–23
1.4.3.2 Supplying an Alternative Version of SYS$LIBRARY 1–24

2 Migrating to the Compaq C Compiler

2.1 General Porting Approach 2–1
2.2 Coexistence of VAX C & Compaq C 2–1
2.3 Compiler Mode and Message Control 2–2
2.3.1 Choosing the Compilation Mode—The /STANDARD Qualifier 2–2
2.3.2 Using Compiler Message Control—/WARNINGS, #pragma message 2–3
2.3.2.1 The /WARNINGS Qualifier 2–4
2.3.2.2 The #pragma message Preprocessor Directive 2–5
2.4 Using Compiler Modes and Message Control to Port your Program 2–6
2.5 New Mechanism for Controlling External Objects 2–7
2.5.1 Extern Model Defaults Have Changed 2–7
2.6 Mixing VAX C and Compaq C Object Files 2–8
2.7 Using Predefined Macros to Aid Coexistence of *VAX C* and *Compaq C* code 2–9
2.8 Additional Information 2–11

Tables

1 Conventions Used in this Guide vii
1–1 Changes in printf Behavior Required by ANSI 1–7
1–2 Linking Conflicts 1–20
2–1 *Compaq C* Compiler Modes 2–2
2–2 /WARNINGS Qualifier Options 2–4

Preface

Purpose of this Guide

This guide is provided as an aid to porting *VAX C* applications to *Compaq C* on *OpenVMS VAX* systems. Chapter 1 describes how to migrate your *VAX C* applications from the *VAX C* Run-Time Library (RTL) to the *Compaq C* RTL for their run-time operations. Chapter 2 describes techniques for helping *VAX C* developers migrate to the *Compaq C* compiler.

This guide is not intended as a summary of all differences between *Compaq C* and *VAX C*. For such a summary, see the New and Changed Features appendix of the *Compaq C User's Guide for OpenVMS Systems*. This appendix lists the features that distinguish *Compaq C* for *OpenVMS* Systems from *VAX C* Version 3.2.

You may send comments or suggestions regarding this guide, or any *Compaq C* document, by electronic mail to the following Internet address:

c_docs@zko.dec.com

Intended Audience

This guide is intended as an aid to *OpenVMS VAX* application developers and users who would like to port their *VAX C* programs to *Compaq C*.

Structure of this Document

This guide is organized as follows:

- Chapter 1 describes how to migrate from the *VAX C* Run-Time Library (RTL) to the *Compaq C* RTL.
- Chapter 2 describes how to port your *VAX C* applications to *Compaq C*.

Associated Documents

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

- *Compaq C Installation Guide for OpenVMS VAX Systems*—For *OpenVMS* system programmers who install the *Compaq C* software on *VAX* systems.
- *Compaq C User's Guide for OpenVMS Systems*—Provides C programmers with information on using *Compaq C* for *OpenVMS* Systems. This guide contains a New and Changed Features appendix that you might find useful. This appendix summarizes the features that distinguish *Compaq C* for *OpenVMS* Systems from *VAX C* Version 3.2.
- *Compaq C Language Reference Manual*—Provides language reference information for *Compaq C* on Compaq systems.
- *Compaq C Run-Time Library Reference Manual for OpenVMS Systems*—Provides information on using the *Compaq C* RTL functions and macros, and information about porting programs to and from other operating systems.
- *The C Programming Language*[1]—Provides an excellent tutorial of the ANSI Standard C language.
- Release Notes—Please read the following release notes for additional information and last-minute bulletins about *Compaq C* and the *Compaq C* RTL:
  - *Compaq C* Version 6.2 for *OpenVMS VAX* Release Notes
  - Release Notes for your version of *OpenVMS VAX*
  - *Compaq C/C++* Run-Time Components for *OpenVMS VAX* Version 6.0-1 Release Notes

Conventions

Table 1 lists the conventions used in this guide.

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

Table 1 Conventions Used in this Guide

Convention	Description
UPPERCASE TEXT	Uppercase letters indicate the name of a command, a file, a parameter, a procedure, or utility.
$ **@SYS$UPDATE:VMSINSTAL**	Monospace boldface text represents user input in interactive examples.
$	The dollar sign is used to indicate the DCL prompt. This prompt may be different on your system.
Ctrl/x	In procedures, a sequence such as Ctrl/X indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
. . .	Vertical ellipsis in an example means that information not directly related to the example has been omitted.
file-spec, . . .	Horizontal ellipsis indicates that additional parameters, values, or information can be entered.
[ ]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)

1

Migrating to the Compaq C Run-Time Library

This chapter describes why and how to migrate your *VAX C* applications to use the *Compaq C* Run-Time Library (RTL) for their run-time operations. It includes several ways to link *Compaq C* and *VAX C* code with the *Compaq C* RTL, descriptions of potential migration concerns and solutions, and interoperability concerns when using multiple C RTLs.

1.1 Why Migrate to the Compaq C RTL

Compaq C introduces a new run-time library (RTL) on *OpenVMS VAX* systems to supplement the original *VAX C* RTL. The new *Compaq C* RTL is designed to be faster and more reliable and maintainable than the *VAX C* RTL, and to support the new ANSI C conformant *Compaq C* compiler for *OpenVMS VAX* systems, as well as the *Compaq C++* compiler for *OpenVMS VAX* systems.

Both the *VAX C* RTL and the *Compaq C* RTL can coexist on your *OpenVMS VAX* system. The *VAX C* RTL supports existing *VAX C* applications. The *Compaq C* RTL supports ANSI-compliant *Compaq C* and *Compaq C++* applications, as well as other components of the *OpenVMS* environment. You can, however, link your *VAX C* programs to the *Compaq C* RTL, as described in the next section.

There are several reasons to port your *VAX C* application's run-time support to the *Compaq C* RTL:

- The *VAX C* RTL will remain unchanged from its Version 5.5-2 release.
- New features, functions, and bug fixes are available only with the *Compaq C* RTL. Some of these include:
  - ANSI C Standard conformance
  - Improved performance
  - Support for DECthreads

  - Improved routine reentrancy
  - Resolution of many *VAX C* RTL bugs

- Using the *Compaq C* RTL provides better portability. Because the *Compaq C* RTL conforms to the ANSI C Standard, you can develop ANSI C conformant programs that can be ported to any other platform that has an ANSI C conformant environment. Also, the *Compaq C* RTL on *OpenVMS VAX* systems is almost the same as the *Compaq C* RTL on *OpenVMS Alpha* systems. Therefore, a project that ports to *OpenVMS Alpha* is better able to have a common code base if it migrates to the *Compaq C* RTL on *OpenVMS VAX* systems.

You would not want to port a *VAX C* application's run-time support to the *Compaq C* RTL for either of the following reasons:

- The application is not under active development.
- The application is tied to *VAX C* behavior that is incompatible with *Compaq C* behavior.

1.2 How to Migrate to the Compaq C RTL

There are three ways to migrate a C application's run-time support from the *VAX C* RTL environment to the *Compaq C* RTL:

- Migrating using the *Compaq C* compiler (Section 1.2.1)
- Migrating *VAX C* code using shareable images (Section 1.2.2)
- Migrating *VAX C* code using object libraries (Section 1.2.3)

The method you choose depends on whether your application source code can be changed, or whether the project distributes shareable images.

1.2.1 Migrating Using the Compaq C Compiler

The *Compaq C* RTL is primarily designed to be used with the *Compaq C* and *Compaq C++* compilers. A program developed with either of these compilers can be linked directly with the *Compaq C* RTL shareable image DECC$SHR.EXE because the compiler generates the correct symbol names and types to be able to link with DECC$SHR.EXE.

The *VAX C* compiler cannot generate code that can be linked directly to DECC$SHR.EXE (although, as discussed in Section 1.2.2, *VAX C* code can be indirectly linked to the *Compaq C* shareable image using the transfer vectors in VAXC2DECC.EXE and VAXCG2DECC.EXE). The only supported way to link directly to DECC$SHR.EXE is to use the *Compaq C* or *Compaq C++* compilers.

By linking directly to DECC$SHR.EXE, the migrating project avoids a number of potential interoperability problems (see Section 1.4).

You can also link your *Compaq C* or *Compaq C++* code to the *Compaq C* RTL object libraries.

Using the *Compaq C* compiler to migrate to the *Compaq C* RTL provides the greatest benefits to the migrating project: easier porting to *OpenVMS Alpha* and *Tru64 UNIX* systems, better error checking, bug fixes, faster and better code.

1.2.1.1 Linking with the Compaq C RTL Shareable Images

Most linking needs should be satisfied by using the *Compaq C* RTL shareable image DECC$SHR.EXE in the SYS$LIBRARY directory. Use this linking method for programs that are written entirely in *Compaq C* or *Compaq C++* code; that is, with no *VAX C* object modules.

Because DECC$SHR.EXE exports only prefixed universal symbols (ones that begin with DECC$), to successfully link against it make sure you cause prefixing to occur for all *Compaq C* RTL entry points.

If you use only the *Compaq C* RTL functions defined in the ANSI C Standard, all entry points will be prefixed.

If you use *Compaq C* RTL functions not defined in the ANSI C Standard, you must compile in one of two ways to ensure prefixing:

- Compile with the /PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES qualifier.
- Compile with the /STANDARD=VAXC or /STANDARD=COMMON qualifier; you get /PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES as the default.

Then link against the shareable image using the standard LINK command. For example:

```
$ LINK PROG1
```

Unlike *VAX C*, *Compaq C* does not require LNK$LIBRARY logical names or options files.

1.2.1.2 Linking with the Compaq C RTL Object Libraries

You can link *Compaq C* object modules to the *Compaq C* RTL object libraries in the same way that you link *VAX C* object modules to the *Compaq C* RTL object libraries. See Section 1.2.3.

1.2.2 Migrating VAX C Code Using Shareable Images

Programs currently linked against the *VAX C* RTL shareable images can be most conveniently migrated to the *Compaq C* RTL by avoiding recompilation entirely and linking existing *VAX C* generated object files with the following shareable images:

- VAXC2DECC.EXE
- VAXCG2DECC.EXE

VAXC2DECC.EXE provides the same transfer vectors and global symbols as VAXCRTL.EXE. VAXCG2DECC.EXE corresponds to VAXCRTLG.EXE. The VAXC2DECC images contain very little code; they consist primarily of transfer vectors into the *Compaq C* RTL shareable image DECC$SHR.EXE.

To use either of these images, replace the current *VAX C* RTL shareable image in your linker options file with its corresponding VAXC2DECC image. For example, a program that used to be linked with VAXCRTL.EXE can be migrated to the *Compaq C* RTL by linking as follows:

```
$ LINK/EXEC=MYPROG MYMOD1,MYMOD2,SYS$INPUT:/OPTIONS
SYS$LIBRARY:VAXC2DECC/SHARE
Ctrl/Z
```

Because the VAXC2DECC images provide the same application program interface (API) as the VAXCRTL images, they have the same potential for universal symbol and psect name conflicts as the VAXCRTL images. See Section 1.4.2 for more information.

1.2.3 Migrating VAX C Code Using Object Libraries

Programs currently linked against the *VAX C* RTL object libraries can be migrated to the *Compaq C* RTL by linking them with the *Compaq C* RTL object libraries. The *Compaq C* RTL object libraries provide both the *VAX C* RTL and *Compaq C* RTL APIs.

The *Compaq C* RTL object libraries reside in the SYS$LIBRARY directory, except they use "DECC" in their name instead of "VAXC":

- DECCRTL.OLB replaces VAXCRTL.OLB
- DECCRTLG.OLB replaces VAXCRTLG.OLB
- DECCCURSE.OLB replaces VAXCCURSE.OLB

Use them just like you would use the *VAX C* RTL object libraries. For example, a program that uses the Curses package and links against the *VAX C* RTL object libraries could use the *Compaq C* RTL by linking as follows:

```
$ LINK/EXEC=MYPROG MYMOD1,MYMOD2, -
SYS$LIBRARY:DECCCURSE/LIBRARY,SYS$LIBRARY:DECCRTL/LIBRARY
```

Note that, as with *VAX C*, if you specify more than one object library on the LINK command, you must do so in the following order:

1. DECCCURSE.OLB
2. DECCRTLG.OLB
3. DECCRTL.OLB

Notes

- Use care if you define any LNK$LIBRARY logicals to the *Compaq C* RTL libraries because doing so will cause all C development (both *VAX C* and *Compaq C*) to use the *Compaq C* object libraries.
- When linking /NOSYSSHR with the *Compaq C* RTL object libraries, you must also specify /INCLUDE=CMA$TIS. See Section 1.3 for more information.
- Because the *Compaq C* RTL object libraries provide the same API as the *VAX C* RTL, they have the same potential for universal symbol and psect name conflicts as the *VAX C* RTL object libraries. See Section 1.4.2 for more information.

1.3 Potential Migration Concerns and Solutions

This section describes migration problems you might encounter when linking /NOSYSSHR with the *Compaq C* RTL object libraries, or because of differences in the behavior of C RTL routines, obsolete routines, and debugging differences.

Also see Section 1.4 for a description of interoperability concerns when multiple copies of the C RTL are used in the same process.

1.3.1 Linking /NOSYSSHR with the Compaq C RTL Object Libraries

If you need to link your program with the *Compaq C* RTL object libraries and /NOSYSSHR, or if want to create an image that uses only *Compaq C* RTL object libraries, you must specify /INCLUDE=CMA$TIS with the object library on the LINK command. Otherwise, the following symbols will be undefined and the resulting image will not execute:

```
%LINK-I-UDFSYM,          CMA$DECC$G_REENTRANCY
%LINK-I-UDFSYM,          CMA$TIS_ERRNO_GET_ADDR
%LINK-I-UDFSYM,          CMA$TIS_ERRNO_SET_ADDR
%LINK-I-UDFSYM,          CMA$TIS_ERRNO_SET_VALUE
%LINK-I-UDFSYM,          CMA$TIS_VMSERRNO_GET_ADDR
%LINK-I-UDFSYM,          CMA$TIS_VMSERRNO_SET_ADDR
```

Here is an example of linking /NOSYSSHR to the *Compaq C* RTL object library:

```
$ LINK/NOSYSSHR PROG1,SYS$LIBRARY:DECCRTL.OLB/LIBRARY/INCLUDE=CMA$TIS
```

1.3.2 Behavior Differences Between the VAX C RTL and Compaq C RTL

This section describes differences in behavior between the *Compaq C* RTL and the *VAX C* RTL. Unless specifically mentioned otherwise, each difference was introduced because of the need to comply with the ANSI C Standard:

- The ANSI C Standard-defined behavior for negated scansets in the `scanf` function is different from the *VAX C* RTL behavior. The ANSI C Standard requires at least one character to match a negated scanset; the *VAX C* RTL allowed no characters to match. For example, the following code fails if it encounters an empty line:

  ```
  /* read lines and throw away the newline */
  fscanf("%[^\n]%*c", &string);
  ```

  The *Compaq C* RTL requires the code to be implemented as follows:

  ```
  /* read lines and throw away the newline */
  if (fscanf("%[^\n]%*c", &string) == 0 )
      fscanf("%*c");  /* swallow blank-line newline */
  ```

- The ANSI C Standard padding rules for the `printf` routines are different than the *VAX C* RTL behavior. Table 1–1 shows examples of the new and old behavior.
- The ANSI C Standard rules for the output of floating-point numbers is different from the (*VAX C* RTL behavior, so that a digit of precision can be lost. Table 1–1 shows examples of the old and new behavior.

- The ANSI C Standard requires that an empty precision value in a `printf` format specifier be treated as a zero precision. The *VAX C* RTL ignores an empty precision value. Table 1–1 shows an example of the new behavior.

Table 1–1 Changes in printf Behavior Required by ANSI

Example	*VAX C* RTL	*Compaq C* RTL
`printf("%04.2d", 77)`	`"0077"`	`"  77"`
`printf("%6.4d", 77)`	`"    77"`	`"  0077"`
`printf("%.2g", 9.876e+2)`	`"9.88e+02"`	`"9.9e+02"`
`printf("%-.s", "hello")`	`"hello"`	`""`

- Exit handlers established by `atexit` are not executed when `abort` is called.
- Files containing binary data that are opened with stream I/O must specify `"ctx=bin"` on the `open`, `fopen`, or `creat` call. (This is a bug fix, not an ANSI C requirement.)
- Files opened in append mode always have writes forced to End-of-File (EOF). With the *VAX C* RTL, the behavior of files opened in append mode was to start at EOF and, thereafter, write at the current file position.
- Files opened for write access default to exclusive file sharing mode. With *VAX C*, files are opened with the `"shr=get"` option set by default.
- A call to `ungetc` clears the EOF flag for the file.
- A call to a file position routine (for example, `fseek`) clears the EOF flag for the file.
- The EOF flag for a file is always cleared when the file is first opened. The *VAX C* RTL sets the EOF flag to true if the file opened was empty.
- A call to a file position routine (for example, `fseek`) now causes the character pushed back by a call to `ungetc` to be lost.
- The C signal SIGABRT is defined to be equal to the SIGIOT signal rather than to SIGILL as in the *VAX C* RTL. Programs that look for the SS$_OPCDEC exception to prevent an `abort` from occurring will fail. The exception to look for is now SS$_OPCCUS.

- The value of the *Compaq C* RTL constant BUFSIZ is 8192, compared with 512 for *VAX C*. This changes the behavior of `setvbuf` and `setbuf` for both *Compaq C* and *VAX C*:
  - `setvbuf` ignores buffers smaller than BUFSIZ. This results in the user's buffer not being used; the previous buffer or default buffer gets used.
  - A program that links with the *Compaq C* RTL and calls `setbuf` with a pointer to a buffer smaller than 8192 bytes will not execute properly because the *Compaq C* RTL writes past the end of the buffer. *VAX C* programs that link to the *Compaq C* RTL using VAXC2DECC.EXE or VAXCG2DECC.EXE will execute properly but will not benefit from the larger buffer size.

  This change was made to improve *Compaq C* RTL I/O performance.
- The `gets` and `fgets` functions no longer null-terminate the input buffer when EOF is encountered and no characters have been read since the last read operation.
- The *Compaq C* RTL I/O functions allow the writing of 0-length records. Passing a length of 0 to `write` and `fwrite` now results in an empty record; the *VAX C* RTL would do nothing. This change was requested by users.
- The `signal` function now returns SIG_ERR on error. This should not affect the behavior of programs that check for –1.
- If the parameter to `fflush` is NULL, all buffers associated with all currently open files are flushed.
- File positioning (for example, `fseek`) is now allowed on files opened in append mode.
- `ftell` now reports a file position that takes into account the character pushed back by a call to `ungetc`.
- The *Compaq C* RTL will not accept uppercase versions of `printf` and `scanf` format specifiers. That is, the *Compaq C* RTL will not accept "%D" as a format specifier, whereas the *VAX C* RTL accepts "%D" as a synonym for the "%d" format specifier. The *Compaq C* RTL prints the letter encountered; for example, "%D" produces the letter 'D' in the output stream rather than being processed as an output source.
- The *Compaq C* RTL allows file positioning calls to seek to an arbitrary byte with fixed-length record files. The *VAX C* RTL treats fixed-length record files like all record files and only allows file positioning to record boundaries. This change was requested by users.

- In *Compaq C*, the `sleep` function wakes prematurely if a signal is received.

1.3.3 Compaq C RTL Obsolete Features

This section lists features that are considered obsolete in the *Compaq C* RTL; they are retained for *VAX C* RTL compatibility only and should not be used for any new development.

1.3.3.1 VAXC$MALLOC_OPT and Related Routines

The following routines are considered obsolete in the *Compaq C* RTL. The *Compaq C* RTL versions of the standard C memory allocation routines are faster and more efficient than the *VAX C* RTL versions.

- VAXC$CALLOC_OPT
- VAXC$CFREE_OPT
- VAXC$FREE_OPT
- VAXC$MALLOC_OPT
- VAXC$REALLOC_OPT

1.3.3.2 sys_nerr and sys_nerrlist Global Symbols

The global symbols sys_nerr and sys_errlist are obsolete in the *Compaq C* RTL. They are provided for *VAX C* RTL compatibility only; new error messages have not been added for a number of releases of the *VAX C* RTL. The symbols are still provided in the `<perror.h>` header file. Programmers are advised to use the portable ANSI C functions `perror` or `strerror` instead.

1.3.4 Debugging and the Compaq C RTL Object Library

The *Compaq C* RTL object library contains a LIB$INITIALIZE psect contribution. When debugging programs linked against the *Compaq C* RTL versions of the object libraries, you must enter GO to reach the main program, and error tracebacks will have an additional call stack listed. For example:

```
$ CC/DECC/DEBUG/NOOPTIMIZE HELLOWORLD
$ LINK/DEBUG HELLOWORLD,SYS$LIBRARY:DECCRTL/LIBRARY
$ RUN HELLOWORLD

         VAX DEBUG Version V6.0DS-004

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

```
DBG> GO
break at routine HELLOWORLD\main
   425: main()
DBG> GO
Hello world!
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG> EXIT
$
```

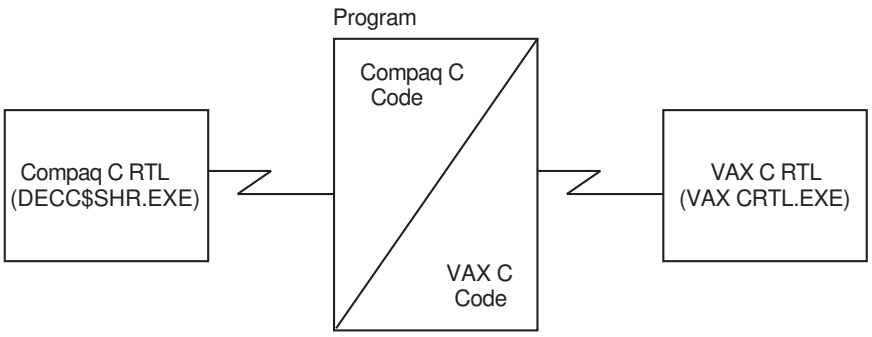
1.4 C RTL Interoperability Concerns

This section documents interoperability restrictions that apply when multiple copies of the C RTL are used in the same process (or in spawned processes) as part of running an application.

Most applications are expected to work correctly with any mix of C RTLs in the same process. However, any shareable image that exchanges part of the C RTL context with its caller could lead to problems in an image that uses two or more C RTLs (see Section 1.4.1). Also, there can be link conflicts when building an application that uses multiple C RTLs (see Section 1.4.2). Finally, you must be sure to avoid version mismatches of RTL components when distributing certain Compaq or user-written applications to other systems (see Section 1.4.3).

The following are examples of mixed C RTL use:

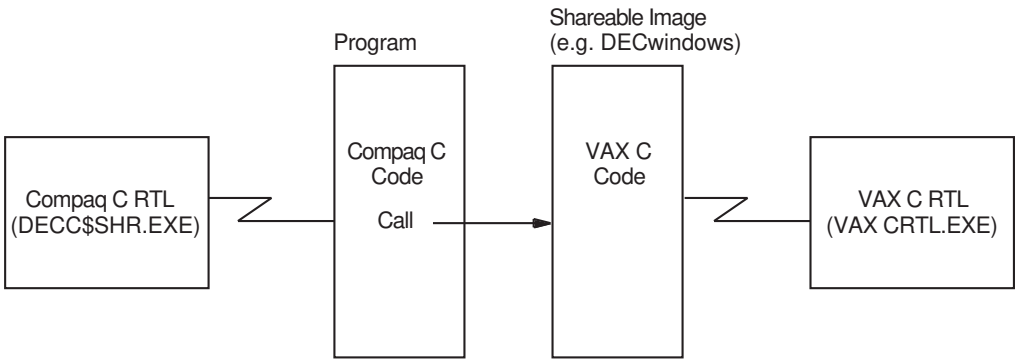
- A single program is composed of modules created by both *VAX C* and *Compaq C* (or *Compaq C++*) and is linked in the default manner so that the *VAX C* modules call VAXCRTL.EXE and the *Compaq C* modules call DECC$SHR.EXE:



ZK–7172A–GE

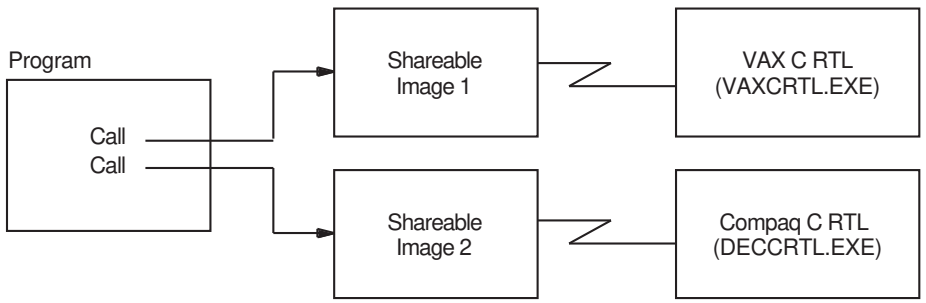
- A program is linked with DECC$SHR.EXE (because it was created with *Compaq C* or *Compaq C++* or it was linked using DECCRTL.OLB), and

also calls some shareable image that is linked with VAXCRTL.EXE, such as DECwindows, CDD, and so forth:



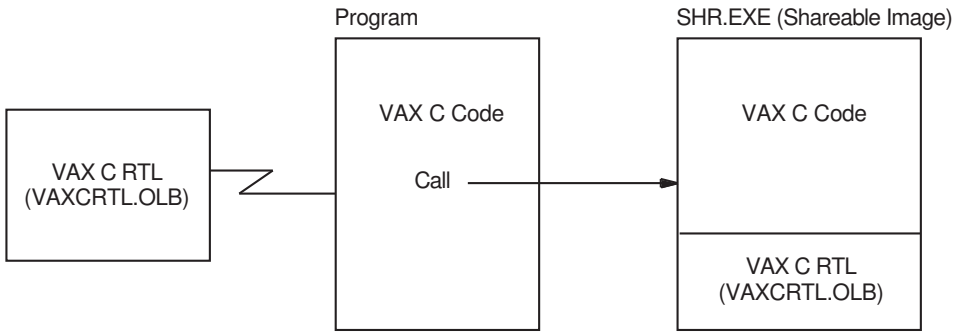
ZK–7173A–GE

- A program calls two different shareable images. One shareable image links with VAXCRTL.EXE and the other links with DECC$SHR.EXE:



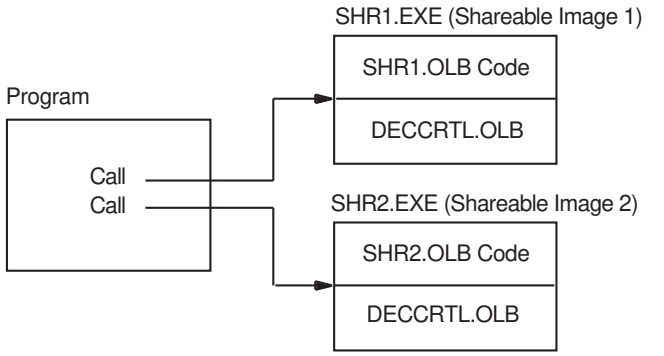
ZK–7169A–GE

- A program is linked with VAXCRTL.OLB and also calls some shareable image that is linked with VAXCRTL.OLB or DECCRTL.OLB:



ZK–7170A–GE

- A program calls two different shareable images. Each shareable image was linked with DECCRTL.OLB:



ZK–7171A–GE

1.4.1 Problems with Images that Export C RTL Context

Most applications are expected to work correctly with any mix of C RTLs in the same process. However, any shareable image that exchanges part of the C RTL context with its caller could lead to problems in an image that uses two C RTLs. Furthermore, these problems can occur in an indeterminate way.

The term *C RTL context* refers to the program state information that is maintained by the C RTL to implement C language semantics. Generally, this is information that applies to the program as a whole, not to one specific subroutine. An example of C RTL context is the file descriptor table. File descriptors are a C language concept maintained by the C RTL. Application variables and the process priority are examples of what is not part of C RTL

context. The first is part of the application context. The second is part of the *OpenVMS* context.

A program that exports C RTL context is said to be *unencapsulated*; one that does not export C RTL context is said to be *encapsulated*.

An exporter of C RTL context is likely to be a programming library, but any program consisting of multiple shareable images might also pass C RTL context between images.

Exporting C RTL context between two instances of C RTLs generally will not work. A shareable image that needs to pass C RTL context to its caller must ensure that the caller will be built with the same copy of a C RTL as the shareable image. C RTL context can never be passed between a C RTL object library and a C RTL shareable image, even if they are instances of the same C RTL.

The following is a complete list of the data structures and information that comprise the C RTL context:

- The free memory list maintained by `malloc` and `free`
- File descriptors and file pointers, including `stdin`, `stdout`, and `stderr`
- `errno`
- Child process context
- Random number seed
- Signal context
- `VAXC$CRTL_INIT` and `main` context, including the `environ` variable

There are some general solutions to these restrictions. The provider of a subsystem has a couple of options:

- If the subsystem appears as an extension to the C language environment, one solution is to build a different version of the subsystem for each of the C environments.
- If the subsystem is an application program interface (API) that is independent of C environments, another option is to move usage of the C RTL inside the subsystem API.

For users of an unencapsulated subsystem, one option is to carefully use `#define` preprocessor macros to map appropriate *VAX C* RTL routine calls into *Compaq C* routine calls without mapping all of them (see Section 1.4.1.1).

The following list gives some instances of exporting C RTL context. The term *subsystem* refers to a shareable image that provides a set of routines to be called by the *caller*, which is typically a `main` program, or possibly another shareable image. If any of these instances occur in an application where a subsystem is linked with one C RTL, and the caller of that subsystem is linked with a different C RTL, then the application may not function correctly.

- A subsystem that allocates memory with calls to `malloc` and requires its caller to deallocate the memory at a later time by calling `free` directly. Memory leakage will occur as a result of such an operation.
- A subsystem that opens a file and returns a file pointer for its callers to use by calling `fprintf` directly.
- A subsystem that reads a line from `stdin` and expects its caller to access the read data with calls to `getc`.
- A subsystem that expects its caller to provide a readable value in the `errno` global variable.
- A subsystem that calls `vfork` and expects its callers to perform a corresponding `exec` call.
- Any attempt to `exec` an image that was built with a different C RTL than the image that performed the `exec` call. Possible symptoms of such an attempt would include no output produced by the child process, or a program that fails with a "bad file number" message.
- A subsystem that expects single-character output to the terminal to occur sequentially, in a readable form, when the caller is also performing single-character output to the terminal.

In general, it is a good idea for subsystems to not export C RTL context. If a subsystem used by an application does export C RTL context, the only solution that insures that the application will run correctly is for the subsystem and the caller to use the same copy of that C RTL. For example, if the subsystem is linked with VAXCRTL.EXE, then the caller must also be linked with VAXCRTL.EXE. If the subsystem is linked with DECC$SHR.EXE, then the caller must also be linked with DECC$SHR.EXE. Note that any shareable image that does not export C RTL context will continue to function correctly with whatever version of the C RTL it is currently using.

The following sections describe solutions to the unencapsulated problem.

1.4.1.1 Becoming Encapsulated—Solutions to the General Problem

There are several options available for changing an unencapsulated image into an encapsulated one:

- The subsystem developer that provides the API definition can change the subsystem.

  The simplest general solution is to move use of the C RTL inside the subsystem API. This has the advantage of permitting the subsystem to be reimplemented without affecting users and permits the subsystem to be used from programming languages other than C. An example where this was done is the X Windows Intrinsics function `XtFree`. Callers of `XtMalloc` previously released virtual memory using the C RTL `free` function. Now users must call `XtFree`.

- The application programmer that uses the subsystem can write a jacket around the unencapsulated API.

  Suppose subsystem XYZ provided a routine `XYZCreate` that returned a pointer to virtual memory, and the application released that virtual memory using `free`. The application could create a routine `free_for_XYZ` that calls `free`. By using `free_for_XYZ` with the same C RTL (either *Compaq C* or *VAX C*) used by XYZ, the application can encapsulate subsystem XYZ. This method is shown in the following example:

```
$ create my_api_memory.c
void free_for_XYZ (void *p)
{
  free (p);
  return;
}
$ CC/VAXC my_api_memory.c
$ LINK/MAP/FULL/CROSS myapp, my_api_memory, SYS$INPUT/OPT
SYS$SHARE:VAXCRTL/SHARE
```

  Note that the jacket routine `my_api_memory.c` is compiled with *VAX C* and linked as for a mixed *Compaq C*/*VAX C* application.

  There are many variations on this theme. Consider a subsystem XYZ that uses *Compaq C* and an application that uses *VAX C*. The application can use mostly *VAX C* code, but use `#define` preprocessor macros to change *VAX C* RTL function calls that would otherwise export C RTL context to function calls to the *Compaq C* RTL. Since all *Compaq C* RTL routine names begin with the DECC$ prefix, a sample `#define` macro might look like this:

```
#define free_for_XYZ decc$free
```

1.4.1.2 Issues for File I/O

If a subsystem returns C file descriptors or file pointers, the application and subsystem must both use the same C RTL. You can use the previous general techniques to encapsulate an unencapsulated subsystem.

For the case of `stdin`, `stdout`, and `stderr`, each instance of the C RTL has its own channel open to these files. The resulting behavior is line-at-a-time processing of input or output. This is much the same as if the application mixed C standard I/O with the standard input or output channels of a different programming language.

1.4.1.3 Considerations for errno

The *Compaq C* RTL is dependent on a new image, SYS$LIBRARY:CMA$TIS_SHR.EXE, to provide `errno` support. If you implement a run-time library that modifies `errno` to communicate error status to callers, do not modify `errno` directly. If you do, the modified `errno` will not be communicated to all C RTLs in the process. For example, if a shareable image that modifies `errno` is linked with the *Compaq C* RTL, and the main program is linked with the *VAX C* RTL, the main program will never see the modified `errno` value from the shareable image.

The recommended way to modify `errno` is to call the function `cma$tis_set_errno_value`. This function has the following prototype:

```
void cma$tis_errno_set_value(int value);
```

For example, consider an image that needs to modify `errno` and has code similar to the following:

```
if (error_situation)
    {
    errno=ERROR_CODE;
    return(error_status);
    }
```

To correctly modify `errno`, change this code to the following:

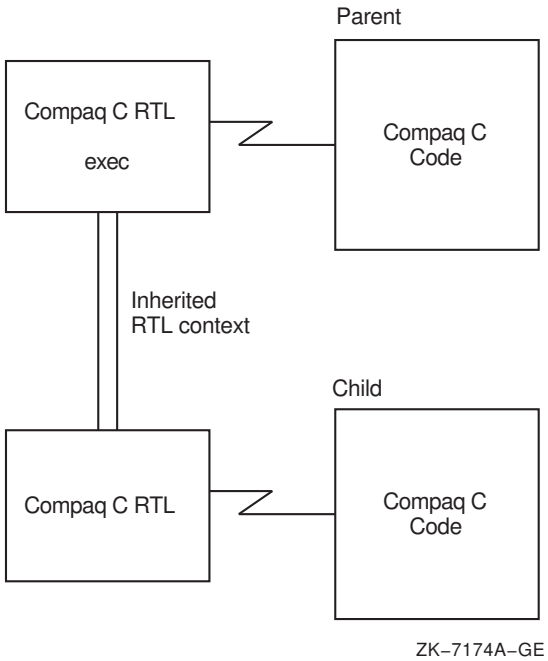
```
if (error_situation)
    {
    void cma$tis_errno_set_value(int);
    cma$tis_errno_set_value(ERROR_CODE);
    return(error_status);
    }
```

Note that programs that merely examine the value of `errno` are not affected by this implementation change.

1.4.1.4 Child Process Context

When spawning a child process using `vfork`, the `main` routine in the child uses a C RTL routine to inherit some context from the parent. This context is documented in the *Compaq C RTL Reference Manual for OpenVMS Systems*, and includes the list of open file descriptors, among other things. The restrictions on mixed C RTL context between different instances of the C RTL applies here as well.

Because of changes in the implementation of forking, both the parent and child process must use the same C RTL to communicate the C RTL context, as shown in the following figure:



The C RTL context passed from the parent is limited to the context visible from the `exec` routine called. Therefore, if the *Compaq C* RTL `exec` routine is called, only files opened in the *Compaq C* RTL are passed to the child process.

In the child process, the C RTL routine called by `main` re-establishes the context that gets used. Therefore, if `main` is using the *Compaq C* RTL, the inherited file descriptors are reopened by the *Compaq C* RTL. These file descriptors are not visible to the *VAX C* RTL.

A related problem is context between vfork and exec. The vfork routine establishes some context that gets used by the exec routines. While this context is implicitly passed (rather than explicitly by the application), it has the same restriction as other mixed C RTL context. The version of vfork must match the version of exec (that is, a *Compaq C* RTL version of vfork must match the *Compaq C* RTL version of exec).

1.4.1.5 Random Number Seed

Each C RTL linked into the application will have its own random number seed. The effect is that each C RTL will have the exact same pseudo-random sequence. If the application intends to call srand to set the random number seed, each seed must be updated.

1.4.1.6 Signal Context

The C RTL internally maintains some context to describe signal handlers that have been enabled. With multiple instances of the C RTL in one process, there can be multiple signal handlers enabled for the same signal. For software signals, the signal handler used is the one that matches the section of the application that raises the signal. For signals caused by hardware exceptions (such as reserved operand faults, or access violations), the signal handler used depends on active call frames. Calls to setjmp establish a C RTL condition handler for the frame. The main program always has a C RTL condition handler enabled. The most recent frame that has a C RTL condition handler established will catch the hardware exception. If this C RTL condition handler is for *Compaq C*, it looks for a signal handler established with a *Compaq C* RTL signal or sigvec routine. If the condition handler is for *VAX C*, then the signal handler established with the *VAX C* RTL gets invoked.

1.4.1.7 VAXC$CRTL_INIT and Context Initialization for main Program

As mentioned previously, some C RTL context initialization occurs when the main program starts and whenever the VAXC$CRTL_INIT routine is called. This initialization includes establishing a *VAX* condition handler to handle hardware exceptions, inheriting context from the parent process if this is a child process, establishing a Ctrl/C handler, setting the default umask, and setting up environment variables accessible through the global object environ.

When multiple C RTLs are used in one application, only one set of context is initialized by default. To get the other C RTL context initialized, you must make an explicit call to the other version of VAXC$CRTL_INIT. If, for example, a main program that is linked with the *VAX C* RTL calls a subroutine that is linked with the *Compaq C* RTL, the subroutine must contain an explicit call to VAXC$CRTL_INIT to initialize the correct RTL context (*Compaq C* RTL, in this case) for later access.

1.4.2 Resolving Link-Time Conflicts with Multiple C RTLs

This section describes the use of interoperability tools to resolve link-time conflicts when using multiple C RTLs.

When migrating to the *Compaq C* RTL, multiple C RTLs will likely be needed to link an application. One C RTL might be explicitly linked against. A second C RTL might not be explicitly linked against, but brought into the link by means of a shareable image. For example, when developing a Motif program using *Compaq C*, the application must be linked against the *Compaq C* RTL and against the Motif images. Motif currently brings the *VAX C* RTL into the link.

Problems encountered when linking with multiple C RTLs are due to the *OpenVMS* linker resolving symbol references in the image being linked by searching the transitive closure of shareable images and libraries. That is, when linking with a shareable image, the linker searches that shareable image and all shareable images referenced in that shareable image. So when linking with VAXCRTL.EXE and with an image linked with VAXCRTLG.EXE, the linker will find two instances of all the C RTL symbols (one in VAXCRTL and one in VAXCRTLG), and report a conflict.

The object libraries do not conflict with routine names, but do conflict with the global symbols. Because *VAX C* implements global symbols as global overlaid psects, the linker attempts to connect all the instances of a C-generated psect with the same name. For example, a reference to `stdin` in the user program is connected with the psect of the same name in VAXCRTL.OLB. However, a shareable image that was linked with VAXCRTL.OLB also has a psect of the same name; this results in an error because the linker cannot connect those two definitions of the psect `stdin`.

Three interoperability tools are provided with the *Compaq C* compiler and in a separate *Compaq C/C++* RTL Run-Time Components kit (see Section 1.4.3.1) to resolve link-time conflicts:

- VAXC$LCL.OPT
- VAXC$EMPTY.EXE
- DECC$EMPTY.EXE

These tools work by hiding the conflicting symbols from one of the C RTLs being linked. Which tool is required depends on what C RTLs are used by the main application and the shareable image.

Table 1–2 shows typical C RTL conflicts and the interoperability tool required to resolve it. In the table, VAXCRTL.EXE refers to either VAXCRTL.EXE or VAXCRTLG.EXE.

Table 1–2 Linking Conflicts

Linker Message	Type of Conflict	Tool Needed
LINK-E-MULSHRPSC	VAXCRTL.OLB/VAXCRTL.EXE	VAXC$LCL.OPT
LINK-E-SHRPSCLNG	VAXCRTL.OLB/DECCRTL.OLB	VAXC$LCL.OPT
LINK-E-MULSHRPSC, LINK-E-SHRPSCLNG	DECCRTL.OLB/VAXCRTL.EXE	VAXC$LCL.OPT
None.	DECCRTL.OLB/DECC$SHR.EXE	DECC$EMPTY
LINK-W-MULDEF	VAXCRTL.EXE/VAXCRTLG.EXE	VAXC$EMPTY
LINK-W-MULDEF	VAXC2DECC.EXE/VAXCRTL.EXE	VAXC$EMPTY

1.4.2.1 Using VAXC$LCL.OPT

VAXC$LCL.OPT is required when building any shareable image linked with the *VAX C* RTL object library or *Compaq C* RTL object library.

If the shareable image is built without using VAXC$LCL.OPT, the C RTL global symbols are visible in the shareable image and cause linker conflicts when users of the image link against it. For example:

```
%LINK-E-MULSHRPSC, psect C$$TRNS_VALUES defined in
        shareable image IMAGE1.EXE; is multiply defined in
        shareable image SYS$LIBRARY:VAXCRTL.EXE;1
-LINK-E-NOIMGFIL, image file not created
```

In this example, the shareable image IMAGE1 uses VAXCRTL.OLB, and the image being linked uses VAXCRTL.EXE. For a successful link, relink the shareable image using VAXC$LCL.OPT:

```
$ LINK/SHARE IMAGE1.OBJ, IMAGE1.OPT/OPTIONS, SYS$LIBRARY:VAXCRTL/LIBRARY, -
_$ SYS$LIBRARY:VAXC$LCL.OPT/OPTIONS
```

The following message also indicates a conflict involving the *VAX C* RTL object library:

```
%LINK-E-SHRPSCLNG, Psect STDIN has length of 8
        in module C$EXTERNDATA file SYS$LIBRARY:DECCRTL.OLB;2
        which exceeds length of 4 in shareable image IMAGE1.EXE;
-LINK-E-NOIMGFIL, image file not created
```

In this example, the shareable image IMAGE1 uses VAXCRTL.OLB, and the image being linked uses DECCRTL.OLB. For a successful link, relink the shareable image using VAXC$LCL.OPT.

If the shareable image cannot be relinked (as in the case of a third-party shareable image), then the interoperability tool can be applied to the main image. If the main image is being linked against DECCRTL.OLB, then apply VAXC$LCL.OPT to the link of the main image.

If the main image is being linked against VAXCRTL.EXE, the only solution is to get the shareable image fixed, because applying any of the interoperability tools to the link of the main image will result in an unsuccessful link.

1.4.2.2 Using VAXC$EMPTY.EXE

Use VAXC$EMPTY.EXE to link a main application with both VAXC2DECC.EXE (or VAXCG2DECC.EXE) and a shareable image linked with VAXCRTL.EXE (or VAXCRTLG.EXE). Using VAXC$EMPTY.EXE hides all the global symbols in the VAXCRTL*.EXE shareable image to prevent conflicts with VAXC2DECC.EXE or VAXCG2DECC.EXE.

Also use VAXC$EMPTY.EXE to link an application with both VAXCRTL.EXE and a shareable image linked with VAXCRTLG.EXE (or vice versa).

When there is a conflict between C RTL shareable images, the linker produces large numbers of messages similar to the following:

```
%LINK-W-MULDEF, symbol ACOS multiply defined
        in module VAXCRTL file SYS$COMMON:[SYSLIB]VAXCRTL.EXE;18
```

In this example, the shareable image is linked with VAXCRTL.EXE, and the main program is linked with VAXC2DECC.EXE.

The solution is to define the VAXCRTL logical to point to VAXC$EMPTY.EXE before linking the main program:

```
$ DEFINE/USER VAXCRTL SYS$LIBRARY:VAXC$EMPTY.EXE
$ LINK/EXEC=MAIN_IMAGE MAIN_PROG,OBJ1,OBJ2,...,SYS$INPUT:/OPTIONS
IMAGE1/SHARE
VAXCRTL/SHARE
Ctrl/Z
```

Note the following about this solution:

- Your linker options file cannot reference SYS$LIBRARY:VAXCRTL; it must reference only VAXCRTL.
- Do not link explicitly against VAXC$EMPTY.EXE, or your application will neither link nor run correctly.
- Do not leave the VAXCRTL pointing to VAXC$EMPTY.EXE, or your application will not run correctly.

- The DEFINE/USER command is used to ensure that the logical definition is removed after execution of the LINK command. Make sure that no commands intervene between the DEFINE/USER command and the LINK command.

Follow the same process when linking against VAXCRTLG.EXE by defining the VAXCRTLG logical to point to VAXC$EMPTY.EXE.

1.4.2.3 Using DECC$EMPTY.EXE

The DECC$EMPTY.EXE interoperability tool allows a program to use the *Compaq C* object library even when the program links with a shareable image that was linked with DECC$SHR.EXE.

If DECC$EMPTY.EXE is not used during the link, all *Compaq C* RTL references from the main program will be resolved in DECC$SHR.EXE, not in the object library. There is no linker message that indicates this fact.

For example, if IMAGE1 is linked against DECC$SHR, and the following link is performed, then the main image will not contain any *Compaq C* RTL object modules. All C RTL references from the main progam are resolved in DECC$SHR:

```
$ LINK/EXEC=MAIN_IMAGE MAIN_PROG,OBJ1,...,SYS$INPUT:/OPTIONS
IMAGE1/SHARE
SYS$LIBRARY:DECCRTL/LIBRARY
Ctrl/Z
```

By defining the DECC$SHR logical to point to DECC$EMPTY.EXE immediately before the link, all references to C RTL symbols from the main program are resolved in the *Compaq C* RTL object library. For example:

```
$ DEFINE/USER DECC$SHR SYS$LIBRARY:DECC$EMPTY.EXE
$ LINK/EXEC=MAIN_IMAGE MAIN_PROG,OBJ1,...,SYS$INPUT:/OPTIONS
IMAGE1/SHARE
SYS$LIBRARY:DECCRTL/LIBRARY
```

Note the following about this solution:

- Do not link explicitly against DECC$EMPTY.EXE, or your application will neither link correctly nor run correctly.
- Do not leave the DECC$SHR logical pointing to DECC$EMPTY.EXE, or your application will not run correctly.
- The DEFINE/USER command is used to ensure that the logical definition is removed after execution of the LINK command. Make sure that no commands intervene between the DEFINE/USER command and the LINK command.

1.4.3 Distributing Programs to Other Systems—Version Dependency

Read this section if you distribute to other *OpenVMS VAX* systems Compaq or user-written *Compaq C* or *Compaq C++* programs, or applications that link either to the *VAX C* RTL or to a shareable image linked with the *VAX C* RTL. An example of the latter is a COBOL program that calls the Common Data Dictionary (CDD); CDDSHR is linked against the *VAX C* RTL.

There are two approaches to solving the problem of distributing such programs to other *OpenVMS VAX* systems without installing a C or C++ compiler product:

- Distribute the *Compaq C/C++* Run Time Components (CRT) kit for installation on the target *OpenVMS VAX* systems. See Section 1.4.3.1.
- Supply development *OpenVMS VAX* systems with an alternative version of SYS$LIBRARY that contains older versions of CRT component files. This eliminates the dependency on the CRT kit. See Section 1.4.3.2.

  This approach is only for programs that link to VAXCRTL.EXE or to other shareable images that link to VAXCRTL.EXE.

1.4.3.1 Using the Compaq C/C++ Run-Time Components Kit

The *Compaq C/C++* Run-Time Components (CRT) kit provides the interoperability tools VAXC$LCL.OPT, VAXC$EMPTY.EXE, DECC$EMPTY.EXE, and other shareable images, message images, and object libraries needed by C and C++ programs compiled with the *Compaq C* and *Compaq C++* compiler products for *OpenVMS VAX* systems at link and run time. Installing this kit on target *OpenVMS VAX* systems allows Compaq and user-written C and C++ programs developed on other *OpenVMS VAX* systems to be run without installing a C or C++ compiler product.

Any images that reference these run-time components will fail to execute if copied to systems where these images have not been provided. This means that if you develop an application with the CRT kit, you must install the CRT kit on every target system where the application will run (unless your application can be distributed using the method described in Section 1.4.3.2, in which case you can choose either method).

For more information on the CRT kit, see the *Compaq C/C++ Run-Time Components for OpenVMS VAX Version 6.0-1 Release Notes*.

1.4.3.2 Supplying an Alternative Version of SYS$LIBRARY

This method is a distribution option for applications that link to VAXCRTL.EXE or to other shareable images that link to VAXCRTL.EXE.

Application developers who want to continue to compile and link programs that are portable to systems where no CRT kit is installed, need to have an alternative version of SYS$LIBRARY containing the older versions of CRT component files. To provide this alternative version, copy all files in SYS$LIBRARY into another directory before you install the CRT kit. After the CRT kit installation, applications that need to link with pre-CRT versions of the shareable images can be supported without change by temporarily defining the logical name SYS$LIBRARY to point to this alternative SYS$LIBRARY before compiling and linking. With this method, the end user of the application sees no change from the previous, unmodified use of SYS$LIBRARY. The temporary definition of SYS$LIBRARY is not needed at run time.

See Chapter 2 of the *Compaq C/C++ Run-Time Components for OpenVMS VAX Reference and Installation Guide* for a more complete discussion of this topic.

The following is an example of saving SYS$LIBRARY. The ANALYZE/IMAGE and SEARCH commands show how to check your program for dependency on the CRT kit. In this example, HELLOWORLD is not dependent:

```
$ CRE/DIR SYS$COMMON:[V55_SYSLIB]
$ COPY SYS$COMMON:[SYSLIB]*.* SYS$COMMON:[V55_SYSLIB]
$ !
$ ! Install Compaq C/C++ Run-time Components Kit
$ !
$ DEFINE SYS$LIBRARY SYS$COMMON:[V55_SYSLIB]
$ CC HELLOWORLD
$ LINK HELLOWORLD, SYS$INPUT:/OPTIONS
SYS$LIBRARY:VAXCRTL/SHARE
Ctrl/Z

$ ANALYZE/IMAGE/OUTPUT=HELLOWORLD HELLOWORLD
%ANALYZE-I-ERRORS, SYS$SYSROOT:[SYSMGR]HELLOWORLD.EXE;2   0 errors
$ SEARCH HELLOWORLD.ANL DECC$SHR,CMA$TIS_SHR
%SEARCH-I-NOMATCHES, no strings matched
$
```

Here is the same example without the DEFINE SYS$LIBRARY. If SEARCH finds either CMA$TIS_SHR or DECC$SHR, as it does in this example, then you need the CRT kit installed on the machine where you run your application:

```
$ DEASSIGN SYS$LIBRARY
$ LINK HELLOWORLD, SYS$INPUT:/OPTIONS
SYS$LIBRARY:VAXCRTL/SHARE
Ctrl/Z
$ ANALYZE/IMAGE/OUTPUT=HELLOWORLD HELLOWORLD
%ANALYZE-I-ERRORS, SYS$SYSROOT:[SYSMGR]HELLOWORLD.EXE;3   0 errors
$ SEARCH HELLOWORLD.ANL DECC$SHR,CMA$TIS_SHR
                        global section name: "CMA$TIS_SHR_001"
               2)  "CMA$TIS_SHR"
$
```

2
Migrating to the Compaq C Compiler

Chapter 1 describes how to migrate your *VAX C* applications from the *VAX C* Run-Time Library (RTL) to the *Compaq C* RTL for their run-time operations.

This chapter describes techniques for helping *VAX C* developers migrate to the *Compaq C* compiler. It discusses some of the *Compaq C* language features available to help you port your applications from VAX C to *Compaq C*, and points out changes in the compiler's default behavior.

2.1 General Porting Approach

Use the *Compaq C* compiler, the *Compaq C* RTL, the /STANDARD=VAXC qualifier, and message control (/WARNINGS qualifier and `#pragma message` preprocessor directive) to ease into the *Compaq C* environment. After you achieve a comfort level compiling in *VAX C* mode, you can then begin to compile with /STANDARD=RELAXED_ANSI to migrate to the more portable, more ANSI-compliant dialect. See Section 2.4 for more details on this approach.

2.2 Coexistence of VAX C & Compaq C

The *VAX C* compiler, header files, and RTL are still available and can coexist with the *Compaq C* compiler on your system.

The CC command is used to invoke either the *VAX C* or *Compaq C* compiler. If your system has a *VAX C* compiler already installed on it, when installing *Compaq C* your system manager specified which compiler the CC command is to invoke by default. To invoke the compiler that is not the default, use the CC command with the appropriate qualifier:

CC/DECC for the *Compaq C* compiler

CC/VAXC for the *VAX C* compiler

To display the default C compiler, enter the following command:

```
$ SHOW LOGICAL DECC$CC_DEFAULT
```

To ensure portability to other *OpenVMS VAX* systems, use CC/DECC explicitly in any command procedures, regardless of what the default is for your system.

2.3 Compiler Mode and Message Control

Compaq C has two complementary qualifiers that control which dialect of C is to be recognized by the compiler, and what messages are generated:

- The /STANDARD qualifier controls the compilation *mode*; that is, what language features and extensions are recognized by the compiler.
- The /[NO]WARNINGS qualifier enables or disables the generation of warning and/or informational messages.

2.3.1 Choosing the Compilation Mode—The /STANDARD Qualifier

The *Compaq C* compiler for *OpenVMS* systems provides several dialects of C. You choose the dialect, or mode, by using the /STANDARD qualifier on the CC command. The /STANDARD qualifier causes the compiler to issue only those warnings appropriate for the dialect of C being compiled. For example, in *VAX C* compatibility mode (/STANDARD=VAXC), the compiler does not issue warnings against *VAX C* extensions; in ANSI C mode, the compiler does issue such warnings.

See Table 2–1 for a description of the *Compaq C* compiler modes.

Table 2–1 *Compaq C* Compiler Modes

Qualifier	Description
/STANDARD=ANSI89	Strict ANSI C mode: Only the ANSI language dialect is recognized. You can use /STANDARD=ANSI89 with /[NO]WARNINGS to control issuance of informational or warning messages. However, the compiler does not recognize many *VAX C* or common C extensions when in strict ANSI C mode (for example, *VAX C* keywords not beginning with two underscores). Therefore, many of the messages normally associated with flagging *VAX C* and common C extensions are not produced.

(continued on next page)

Table 2–1 (Cont.) *Compaq C* Compiler Modes

Qualifier	Description
/NOSTANDARD or /STANDARD=RELAXED_ANSI89 (*default mode*)	Relaxed ANSI C mode: This is the default mode on *OpenVMS VAX* systems. In relaxed ANSI C mode, the compiler follows the ANSI C Standard, but also accepts additional Compaq keywords and predefined macros that do not begin with an underscore.
/STANDARD=VAXC	*VAX C* compatibility mode: This mode of compilation allows the same language as the ANSI C language, but also supports *VAX C* extensions that are incompatible with the ANSI C Standard and that change the language semantics. This mode provides a high degree of compatibility for programs that depend on old *VAX C* behavior. This mode is not a complete emulation of *VAX C*.
/STANDARD=COMMON	Common usage C mode: This mode provides a high degree of compatibility with the `pcc` compiler on ULTRIX systems. This mode is close to a subset of /STANDARD=VAXC mode.
/STANDARD=MIA	MIA conformance mode: This is strict ANSI C with some differences required by the Multivendor Integration Architecture (MIA) standard: • G_floating becomes the default floating-point format for `double` variables. • In structures, 0-length bit fields cause the next bit field to start on an integer boundary, rather than on a character boundary.

Compaq C modules compiled in different modes can be linked and executed together.

2.3.2 Using Compiler Message Control—/WARNINGS, #pragma message

Because CC/DECC/STANDARD=VAXC finds problems that *VAX C* ignores, it issues more warning-level messages than *VAX C* does. Consequently, you can use the /WARNINGS qualifier as a tool to help migrate to /STANDARD=RELAXED_ANSI. You can also use the `#pragma message` preprocessor directive to fine tune compilation to avoid unwanted messages.

Sections 2.3.2.1 and 2.3.2.2 provide reference information about these message-control features.

Section 2.4 describes how to use /STANDARD, /WARNINGS, and `#pragma message` to port your code from *VAX C* to *Compaq C*.

2.3.2.1 The /WARNINGS Qualifier

The /WARNINGS qualifier has the following format:

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

This qualifier controls the issuance of compiler diagnostic messages or groups of messages. The /NOWARNINGS qualifier suppresses the warning and informational messages.

Table 2–2 shows the /WARNING qualifier options.

Table 2–2 /WARNINGS Qualifier Options

Option	Usage
DISABLE=(*message-list*)	Suppresses the issuance of the message identifiers or message group names listed. Only messages of severity Warning (W) or Information (I) can be disabled. If the message has severity of Error (E) or Fatal (F), it is issued regardless of any attempt to disable it.
ENABLE=(*message-list*)	Enables issuance of the message identifiers or message group names listed.
NOINFORMATIONALS	Suppresses informational messages.

The *message-list* can be any one of the following:

- A single message identifier. The message identifier is the name following the severity at the start of a line when a message is issued. For example, in the following message, the message identifier is GLOBALEXT:

```
%CC-W-GLOBALEXT, a storage class of globaldef,
globalref, or globalvalue is a language extension.
```

- The name of a single message group. Message group names are:
  - ALL—All the messages in the compiler
  - CHECK—All the messages about potentially poor coding practices
  - PORTABLE—All the messages about portability
- A single message identifier enclosed in parentheses.

- A single message group name enclosed in parentheses.
- A comma-separated list of message identifiers or group names, freely mixed, enclosed in parentheses.

For a list of all *Compaq C* compiler message identifiers, enter the following command:

```
$ HELP CC/DECC MESSAGES
```

The SUMMARY informational message cannot be suppressed with /NOWARNINGS or /WARNINGS=NOINFORMATIONALS, but it can be suppressed with the DISABLE option.

The default is /WARNINGS. This enables all diagnostic messages for the selected compiler mode except those in the PORTABLE and CHECK groups. The PORTABLE and CHECK groups must be explicitly enabled before their messages can be generated.

2.3.2.2 The #pragma message Preprocessor Directive

The `#pragma message` directive controls the issuance of individual diagnostic messages or groups of messages. Use of this pragma overrides any command-line options that may affect the issuance of messages.

This pragma has the following format:

#pragma message *option1* (*message-list*)
#pragma message *option2*

The parameter *option1* must be one of the following keywords:

- `enable`—Enables issuance of the messages specified in the *message-list*
- `disable`—Disables issuance of the messages specified in the *message-list*

See the /WARNING qualifier (Section 2.3.2.1) for a description of the *message-list*.

Only messages of severity Warning (W) or Information (I) can be disabled. If the message has severity of Error (E) or Fatal (F), it is issued regardless of any attempt to disable it.

The parameter *option2* must be one of the following keywords:

- `save`—Saves the current state of which messages are enabled and disabled
- `restore`—Restores the previous state of which messages are enabled and disabled

The `save` and `restore` options are useful primarily within header files.

2.4 Using Compiler Modes and Message Control to Port your Program

The following steps describe a suggested path for porting your *VAX C* programs to the relaxed ANSI C mode of the *Compaq C* compiler.

1. Use CC/DECC/STANDARD=VAXC, and then do the following:
   a. Make sure the code works.
   b. Consider making code changes on the basis of any diagnostic messages. You must change the code to eliminate E-level diagnostics.
   c. If you are not interested in using ANSI C modes, use `#pragma message` or /WARNINGS to suppress messages you do not want to see.
2. Use CC/DECC/STANDARD=VAXC /WARNINGS=ENABLE=(PORTABLE,CHECK), and then do the following:
   a. Consider making code changes on the basis of any additional diagnostic messages. Some messages might point out language features that work differently between VAXC and ANSI C modes.

      Note that compiling this way just gives you more diagnostics; it does not change the behavior of your code.
   b. If you are not interested in using ANSI C modes, use `#pragma message` or /WARNINGS to suppress warning and informational messages you do not want to see.
3. Use CC/DECC/STANDARD=RELAXED_ANSI, and then do the following:
   a. Consider making code changes on the basis of any additional diagnostic messages.
   b. Make sure the code works. There are language features that behave differently when compiling with /STANDARD=RELAXED_ANSI than when compiling with /STANDARD=VAXC.
   c. If you are not interested in using the strict ANSI C modes (/STANDARD=ANSI89 or /STANDARD=MIA), use `#pragma message` or /WARNINGS to suppress warning and informational messages you do not want to see.

      Stopping at /STANDARD=RELAXED_ANSI is recommended. It becomes increasingly difficult to compile in strict ANSI C mode.
4. Use CC/DECC/STANDARD=ANSI89 or CC/DECC/STANDARD=MIA. This compiles in strict ANSI C mode. This step is not necessary for most *OpenVMS* users and is potentially undesirable.

2.5 New Mechanism for Controlling External Objects

The *Compaq C* compiler provides an alternative, ANSI-compliant method of controlling objects that have external linkage. To take advantage of this method, use the `#pragma extern_model` preprocessor directive and the /EXTERN_MODEL and /[NO]SHARE_GLOBALS command-line qualifiers.

The pragma and command-line qualifiers replace the *VAX C* mode storage-class specifiers (`globaldef`, `globalref`, `globalvalue`) and storage-class modifiers (`noshare` and `readonly`). They allow you to select the implementation model of external data and control the psect usage of your programs. The `_align` macro is still available to ensure object alignment.

For a description of `#pragma extern_model` and its relationship to the external storage classes it replaces, see the Preprocessor Directives chapter of the *Compaq C User's Guide for OpenVMS Systems*.

For a description of the command-line qualifiers /EXTERN_MODEL and /[NO]SHARE_GLOBALS, see the Developing C Programs at the DCL Command Level chapter of the *Compaq C User's Guide for OpenVMS Systems*.

For more information on external storage classes and psects, see the Data Storage chapter of the *Compaq C User's Guide for OpenVMS Systems*.

2.5.1 Extern Model Defaults Have Changed

This section explains how to use /EXTERN_MODEL=COMMON_BLOCK and /SHARE_GLOBALS to obtain the same extern model defaults as *VAX C*.

By default, *VAX C* uses the *common block* model for external objects and treats external objects as *shared* (that is, the psects created for external objects are marked as *shared*). In the common block model, all declarations are definitions, and the linker combines all definitions with the same name into one definition.

The *Compaq C* default for the initial extern model is different from that of *VAX C*, even when compiling in *VAX C* mode: *Compaq C* uses the *relaxed refdef* model and treats external objects as *noshare*. That is, the default qualifiers that govern the initial model for external objects are /EXTERN_MODEL=RELAXED_REFDEF and /NOSHARE_GLOBALS. In the relaxed refdef model, some declarations are references and some are definitions. Multiple uninitialized definitions for the same object are allowed and resolved into one by the linker. However, a reference requires that at least one definition exists. This model is used by C compilers on UNIX systems.

Note that the change in the extern model default from *share* to *noshare* could have the following impact:

- Linking old object files or object libraries with newly produced object files might generate ‘‘conflicting attributes for psect’’ messages. As long as you are not building shareable libraries, you can safely ignore these messages.
- Building shareable libraries will be easier.
- When linking external symbols against FORTRAN common blocks, you should specify /SHARE_GLOBALS to suppress ‘‘conflicting attributes for psect’’ messages, but they can otherwise be ignored.

For compatibility with *VAX C*, you may wish to change the *Compaq C* default extern model. To obtain the same extern model defaults as *VAX C*, use the following qualifiers:

```
$ CC/DECC/EXTERN_MODEL=COMMON_BLOCK/SHARE_GLOBALS
```

2.6 Mixing VAX C and Compaq C Object Files

You can mix object files compiled by the *VAX C* compiler and the *Compaq C* compiler. Because *Compaq C* does not support parallel processing, you might want to mix object files as a workaround for parallel processing usage in your *VAX C* programs.

To mix object files:

1. Separately compile the *VAX C* code and the *Compaq C* code:

   ```
   $ CC/DECC PROG1
   $ CC/VAXC PROG2
   ```

2. Link the object modules together and with the *Compaq C* RTL:

   - To link against the *Compaq C* RTL shareable image, use the /OPTIONS qualifier to specify the VAXC2DECC.EXE shareable image for the *VAX C* compiled file:

     ```
     $ CC/DECC PROG1
     $ CC/VAXC PROG2
     $ LINK PROG1, PROG2, TT:/OPTIONS
     SYS$LIBRARY:VAXC2DECC.EXE/SHARE
     Ctrl/Z
     ```

     Prefixed C RTL calls from PROG1 are resolved in DECC$SHR. Unprefixed C RTL calls from PROG2 are resolved in VAXC2DECC.EXE, which transfers them to DECC$SHR.

- To use the *Compaq C* RTL object libraries:

```
$ CC/DECC PROG1
$ CC/VAXC PROG2
$ LINK PROG1, PROG2, SYS$LIBRARY:DECCRTL.OLB/LIBRARY
```

  All C RTL calls from both PROG1 and PROG2 are resolved in DECCRTL.OLB.

- If you link /NOSYSSHR with the *Compaq C* RTL object libraries, you must also include CMA$TIS:

```
$ CC/DECC PROG1
$ CC/VAXC PROG2
$ LINK/NOSYSSHR PROG1, PROG2, SYS$LIBRARY:DECCRTL.OLB/LIBRARY/INCL=CMA$TIS
```

See Section 1.2 for more information on linking with the *Compaq C* RTL.

2.7 Using Predefined Macros to Aid Coexistence of *VAX C* and *Compaq C* code

This section describes how to use predefined system-identification macros to aid in the coexistence of *VAX C* and *Compaq C* code.

Each implementation of the *Compaq C* compiler automatically defines macros that you can use to identify the system on which the program is running. These macros can assist in writing code that executes conditionally, depending on the architecture or operating system on which the program is running, and on the compiler and/or qualifiers specified.

See the Predefined Macros and Builtin Functions chapter of the *Compaq C User's Guide for OpenVMS Systems* for a list of the traditional (non-ANSI) and new (ANSI) spellings of these predefined macro names for *Compaq C* for *OpenVMS* Systems.

The predefined macro symbols are defined as 1 or 0, as appropriate to the compiler, processor, and compilation qualifiers.

The __DECC Macro

For the purposes of this discussion, we are interested in the __DECC predefined macro. The *Compaq C* compiler always defines __DECC to be 1; it is not defined for the *VAX C* compiler. You can use this fact to separate *VAX C* code from *Compaq C* code, as in the following example:

```
#ifdef   __DECC
#pragma extern_model common_block shr
#endif
```

In this example, if the code is being compiled by *Compaq C*, the default model for external variables is changed from `relaxed_refdef noshr` (the *Compaq C* default) to `common_block shr` (the *VAX C* default). See Section 2.5 for more information about `#pragma extern_model`.

Do not use the `__VAXC` predefined macro to try to distinguish between the two compilers because `__VAXC` is defined to be 1 both by the *VAX C* compiler and by the *Compaq C* compiler in *VAX C* mode.

The __alpha and __Alpha_AXP Macros

Several other predefined macros are available with *Compaq C*. For example, a program can use `__alpha` and `__Alpha_AXP` to perform processor-specific functions on Alpha systems. For example:

```
#ifdef    __alpha
void __PAL_BPT (void);
#endif
```

The __HIDE_FORBIDDEN_NAMES Macro

You might also find the `__HIDE_FORBIDDEN_NAMES` macro useful.

The ANSI C standard specifies exactly what identifiers in the normal name space are declared by the standard header files. A compiler is not free to declare additional identifiers in a header file unless the identifiers follow defined rules (the identifier must begin with an underscore followed by an uppercase letter or another underscore).

When running the *Compaq C* compiler for *OpenVMS* systems in strict ANSI C mode (/STANDARD=ANSI89), versions of the standard header files are included that hide many identifiers that do not follow the rules. The header file `<stdio.h>`, for example, hides the definition of the macro TRUE. The compiler accomplishes this by predefining the macro `__HIDE_FORBIDDEN_NAMES` in strict ANSI C mode.

You can use the command-line qualifier /UNDEFINE="__HIDE_FORBIDDEN_NAMES" to prevent the compiler from predefining this macro and, thereby, including macro definitions of the forbidden names.

The header files are modified to only define additional *VAX C* names if `__HIDE_FORBIDDEN_NAMES` is undefined. For example, `<stdio.h>` might contain the following:

```
#ifndef __HIDE_FORBIDDEN_NAMES
#define TRUE    1
#endif
```

2.8 Additional Information

For additional information on *Compaq C* features, see the *Compaq C User's Guide for OpenVMS Systems*. This guide contains a New and Changed Features appendix that you might find useful. This appendix summarizes the features that distinguish *Compaq C* for *OpenVMS* Systems from *VAX C* Version 3.2. You might also want to read the Common Pitfalls appendix of the user's guide.