
Porting Applications from HP OpenVMS Alpha to HP OpenVMS Industry Standard 64 for Integrity Servers

Order Number: AA-xxxxx-TE

December 2003

This manual provides a framework for application developers who are migrating applications from HP OpenVMS Alpha to HP OpenVMS Industry Standard 64.

Revision/Update Information: This is a new manual.
Software Version: OpenVMS Industry Standard 64 for
Integrity Servers Version 8.1

Hewlett-Packard Company

HP OpenVMS Industry Standard 64
Evaluation Release Version 8.1
is for evaluation purposes only
and is not intended for use
in production environments.

Hewlett-Packard Company
Palo Alto, California

© 2003 Hewlett-Packard Development Company, L.P.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Proprietary computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Intel® and Itanium® are registered trademarks of the Intel Corporation.

UNIX® is a registered trademark of The Open Group.

ZK6673

The HP OpenVMS documentation set is available on CD-ROM.

This document was prepared using DECdocument, Version 3.3-1b.

Contents

Preface	vii
1 Introduction	
1.1 OpenVMS on Industry Standard 64 Integrity Servers	1-1
1.2 Overview of the Porting Process	1-1
1.2.1 Evaluating the Application	1-1
2 Fundamental Differences to Consider	
2.1 Calling Standard	2-1
2.1.1 Changes to the Calling Standard	2-1
2.1.2 Extensions to the Calling Standard	2-2
2.2 Exception Handling	2-3
2.3 Floating-Point Arithmetic	2-3
2.3.1 Overview	2-3
2.3.2 Impact on OpenVMS Applications	2-3
2.4 Object File Format	2-4
3 Assessing Needs for Porting Your Application	
3.1 Overview	3-1
3.2 Evaluating the Application	3-2
3.2.1 Selecting the Porting Method	3-2
3.2.2 Taking Inventory of What Is to Be Ported	3-3
3.2.3 Assessing Dependencies	3-5
3.2.3.1 Software Dependencies	3-5
3.2.3.2 Development Environment	3-6
3.2.3.3 Operating Environment	3-6
3.2.4 Operational Tasks	3-7
3.3 HP Porting Resources	3-8
3.4 Additional Considerations	3-8
3.5 Porting Checklist	3-8
4 Migrating Source Modules	
4.1 Setting Up the Migration Environment	4-2
4.1.1 Hardware	4-2
4.1.2 Software	4-3
4.2 Compiling Applications on Alpha With Current Compiler Version	4-3
4.3 Testing Applications on Alpha for Baseline Information	4-4
4.4 Recompiling and Relinking on an I64 System	4-4
4.5 Debugging the Migrated Application	4-4
4.5.1 Debugging	4-4

4.5.2	Analyzing System Crashes	4-5
4.5.2.1	System Dump Analyzer	4-5
4.5.2.2	Crash Log Utility Extractor	4-5
4.6	Testing the Migrated Application	4-5
4.6.1	Alpha Tests Ported to I64	4-5
4.6.2	New I64 Tests	4-6
4.6.3	Uncovering Latent Bugs	4-6
4.7	Integrating the Migrated Application into a Software System	4-6
4.8	Modifying Certain Types of Code	4-7
4.8.1	Conditionalized Code	4-7
4.8.1.1	MACRO Sources	4-7
4.8.1.2	BLISS Sources	4-8
4.8.1.3	C Sources	4-8
4.8.1.4	Existing Conditionalized Code	4-9
4.8.2	System Services With Alpha Architecture Dependencies	4-9
4.8.2.1	SYS\$GOTO_UNWIND	4-9
4.8.2.2	SYS\$LKWSET and SYS\$LKWSET_64	4-9
4.8.3	Code With Other Dependencies on the Alpha Architecture	4-10
4.8.3.1	Initialized Overlaid Program Sections	4-10
4.8.3.2	Condition Handlers Use of SS\$_HPARITH	4-10
4.8.3.3	Mechanism Array Data Structure	4-10
4.8.3.4	Reliance on Alpha Object File Format	4-10
4.8.4	Code That Uses Floating Point Data Types	4-11
4.8.4.1	LIB\$WAIT Problem and Solution	4-12
4.8.5	Code That Uses Threads	4-13
4.8.5.1	Thread Routines cma_delay and cma_time_get_expiration	4-14
4.8.6	Code With Unaligned Data	4-14
4.8.7	Code That Relies on the OpenVMS Alpha Calling Standard	4-15
4.8.8	Privileged Code	4-15
4.8.8.1	Use of SYS\$LKWSET and SYS\$LKWSET_64	4-15
4.8.8.2	Terminal Drivers	4-16

5 OpenVMS I64 Development Environment

5.1	Native I64 Compilers	5-1
5.1.1	VAX MACRO-32 Compiler for OpenVMS I64	5-1
5.2	Other Development Tools	5-2
5.2.1	Binary Translator	5-3
5.3	Linking Modules	5-3
5.3.1	Differences When Linking on OpenVMS I64 Systems	5-4
5.3.1.1	Mixed-Case Module Names Permitted in Link Operations	5-4
5.3.1.2	Handling of Initialized Overlaid Program Sections on OpenVMS I64	5-4
5.3.1.3	Conventions for Specifying Image Names	5-5
5.3.2	Expanded Map File Information	5-5
5.3.3	New Linker Qualifiers and Options for OpenVMS I64	5-5
5.3.3.1	New /BASE_ADDRESS Qualifier	5-6
5.3.3.2	New Linker Option: SHORT_DATA	5-6
5.3.3.3	New alignments for the PSECT_ATTRIBUTE Option	5-7
5.4	Debugging Capabilities on OpenVMS I64 Systems	5-7

5.4.1	OpenVMS Debugger	5-7
5.4.1.1	Architecture Support	5-8
5.4.1.2	Language Support	5-8
5.4.1.3	Functional Areas and Commands	5-9
5.4.1.4	Functionality Not Yet Ported	5-9
5.4.2	XDelta Debugger	5-9
5.4.2.1	XDelta Capabilities on OpenVMS I64	5-9
5.4.2.2	Differences Between XDelta on OpenVMS I64 and OpenVMS Alpha Systems	5-10
5.5	OpenVMS I64 Librarian Utility	5-10
5.5.1	Considerations When Using the I64 Librarian	5-10
5.5.2	Changes to the LBR\$ Routines	5-11
5.5.3	I64 Library Format Handles Unix-Style Weak Symbols	5-11
5.5.3.1	New ELF Type for Weak Symbols	5-11
5.5.3.2	Version 4.0 Library Index Format	5-11
5.5.3.3	Current Library Limitation with Regard to Weak and Group Symbols	5-12
5.5.3.4	New Group-Section Symbols	5-12

6 Preparing to Port Applications

6.1	Bliss Compiler	6-2
6.1.1	Files	6-2
6.1.1.1	File Types	6-2
6.1.1.2	Output File Location Defaults	6-3
6.1.2	Alpha BLISS Features Not Available	6-3
6.1.2.1	Alpha BLISS Machine-Specific Builtins	6-3
6.1.2.2	Alpha BLISS PALcode Builtin Functions	6-3
6.1.2.3	Alpha BLISS Register Names	6-4
6.1.2.4	Granularity of Byte and Word Stores and Fetches	6-4
6.1.2.5	INTERRUPT and EXCEPTION Linkages	6-4
6.1.2.6	BUILTIN Rn	6-4
6.1.2.7	Link-Time Complex Expressions	6-5
6.1.3	OpenVMS I64 BLISS Features	6-5
6.1.3.1	OpenVMS I64 BLISS Builtins	6-5
6.1.3.1.1	Common BLISS Builtins	6-5
6.1.3.1.2	Machine-Specific Builtins	6-5
6.1.3.1.3	New Machine-Specific Builtins	6-6
6.1.3.1.4	PALcode Builtins	6-7
6.1.3.2	BLISCALLG	6-7
6.1.3.3	I64 Registers	6-7
6.1.3.4	ALPHA_REGISTER_MAPPING Switch	6-8
6.1.3.4.1	/ANNOTATIONS Qualifier	6-9
6.1.3.4.2	/ALPHA_REGISTER_MAPPING Qualifier	6-9
6.1.3.4.3	/ALPHA_REGISTER_MAPPING Informationals	6-9
6.1.3.5	ADD, AND, Builtin Functions for Atomic Operations	6-10
6.1.3.6	TESTBITxxI and TESTBITxx Builtin Functions for Atomic Operations	6-10
6.1.3.7	Granularity of Longword and Quadword Writes	6-10
6.1.3.8	Shift Builtin Functions	6-11
6.1.3.9	Compare and Swap Builtin Functions	6-11
6.1.3.10	I64-Specific Multimedia Instructions	6-11
6.1.3.11	Linkages	6-11
6.1.3.11.1	CALL Linkage	6-11

6.1.3.11.2	JSB Linkage	6-11
6.1.3.12	/[NO]TIE Qualifier	6-11
6.1.3.13	/ENVIRONMENT=([NO]FP) and ENVIRONMENT([NO]FP)	6-12
6.1.3.14	Floating Point Support	6-12
6.1.3.14.1	Floating Point Builtin Functions	6-12
6.1.3.14.2	Floating Point Literals	6-12
6.1.3.14.3	Floating Point Registers	6-12
6.1.3.14.4	Calling Non-BLISS Routines with Floating Point Parameters	6-12
6.1.3.15	New and Expanded Lexicals	6-12
6.1.3.16	OpenVMS I64 BLISS Support for IPF Short Data Sections	6-13
6.2	COBOL	6-15
6.2.1	/ARCH and /OPTIMIZE=TUNE Qualifiers	6-15
6.3	Fortran	6-15
6.3.1	Floating Point Arithmetic	6-15
6.3.2	Only the F90 Compiler is Supported	6-16
6.3.3	/ARCH and /OPTIMIZE=TUNE Qualifiers	6-17
6.4	HP C/ANSI C	6-17
6.4.1	I64 Floating Point Default	6-17
6.4.2	Semantics of /IEEE_MODE Qualifier	6-17
6.4.3	Predefined Macros	6-18
6.5	Java	6-18
6.6	Macro-32	6-18

7 Other Considerations

7.1	Hardware Considerations	7-1
7.1.1	Itanium Processor Family Overview	7-1
7.1.2	Alpha Processor Family Overview	7-1
7.2	Endianism Considerations	7-2

A Application Evaluation Checklist

Glossary

Index

Figures

3-1	Porting an Application	3-4
4-1	Migrating an Application from Alpha to I64	4-1
7-1	Big-Endian Byte Ordering	7-3
7-2	Little-Endian Byte Ordering	7-3

Tables

6-1	Mapping Alpha Compiler Versions to I64 Versions	6-1
-----	---	-----

Preface

Intended Audience

Porting Applications from HP OpenVMS Alpha to HP OpenVMS Industry Standard 64 for Integrity Servers is intended for application developers who are planning to migrate applications from the OpenVMS Alpha operating system to the OpenVMS I64 operating system.

Document Structure

This document contains the following chapters:

Chapter 1 includes an overview of the OpenVMS I64 Version 8.1 operating system, including its evolution and its benefits.

Chapter 2 discusses fundamental differences between the OpenVMS Alpha and OpenVMS I64 operating systems.

Chapter 3 outlines a process for assessing applications in preparation for porting to OpenVMS I64.

Chapter 4 discusses specific tasks that must be completed to prepare source modules for migration, as well as information about compiling, linking, testing, and deploying ported applications.

Chapter 5 discusses the OpenVMS I64 development environment.

Chapter 6 discusses porting considerations related to the primary compilers that will be available for OpenVMS I64.

Chapter 7 discusses other factors to consider when undertaking a porting effort.

Appendix A contains a sample checklist for use when evaluating applications prior to porting.

The Glossary provides a list of terms and their definitions.

Related Documents

Users of this manual may also find the following documents useful:

- *HP OpenVMS Calling Standard*
- *OpenVMS Debugger Manual*
- *OpenVMS Linker Utility Manual*
- *OpenVMS Alpha System Analysis Tools Manual*

Documentation for individual compilers may also be useful in the porting process.

For additional information about HP OpenVMS products and services, visit the following World Wide Web address:

<http://www.hp.com/go/openvms>

Reader's Comments

HP welcomes your comments on this manual. Please send comments to either of the following addresses:

Internet	openvmsdoc@hp.com
Postal Mail	Hewlett-Packard Company OSSG Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How to Order Additional Documentation

For information about how to order additional documentation, visit the following World Wide Web address:

<http://www.hp.com/go/openvms/doc/order>

Conventions

The following conventions may be used in this manual:

Ctrl/x	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.
PF1 x	A sequence such as PF1 x indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.) In the HTML version of this document, this convention appears as brackets, rather than a box.
...	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
.	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
()	In command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one.

[]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold type	Bold type represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
<i>italic type</i>	Italic type indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error <i>number</i>), in command lines (/PRODUCER= <i>name</i>), and in command parameters in text (where <i>dd</i> represents the predefined code for the device type).
UPPERCASE TYPE	Uppercase type indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
Example	This typeface indicates code examples, command examples, and interactive screen displays. In text, this type also identifies URLs, UNIX commands and pathnames, PC-based commands and folders, and certain elements of the C programming language.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radices—binary, octal, or hexadecimal—are explicitly indicated.

Introduction

This chapter contains an introduction to OpenVMS on Industry Standard 64 systems and an overview of the migration process.

For more information, see the Release Notes.

1.1 OpenVMS on Industry Standard 64 Integrity Servers

The OpenVMS I64 architecture has a 64-bit model and basic system functions similar to the Alpha architecture; thus the great majority of applications running on OpenVMS Alpha today can be easily ported to the I64 architecture with few changes.

The OpenVMS Alpha and OpenVMS I64 architecture variants of OpenVMS are produced from a single source code base; thus, non-hardware dependent features may be incorporated into both versions without multiple changes to the source code, minimizing the time required to perform qualification testing, and helping to ensure the availability of critical applications on both OpenVMS platforms.

HP intends to maintain a strict policy of ensuring and maintaining forward source compatibility so that "well-behaved" applications that currently run on recommended versions of OpenVMS Alpha run successfully on OpenVMS I64 systems. If an application takes advantage of published system services and library interfaces, there should be a high level of confidence that the application will move without modifications to the latest version of OpenVMS I64.

OpenVMS I64 has the same look and feel that OpenVMS customers are familiar with. Minor changes were needed to accommodate the new architecture, but the basic structure and capabilities of OpenVMS are the same.

1.2 Overview of the Porting Process

This section contains a high-level overview of the porting process as it will apply to most applications.

1.2.1 Evaluating the Application

Evaluating the application identifies the steps necessary for its migration to HP OpenVMS I64. The result of the evaluation should be a migration plan that answers the following questions:

- How to migrate the application
- How much time, effort, and cost the migration will require
- Whether you require support from HP services

The evaluation process has the following three stages

1. General inventory, including identifying dependencies on other software
2. Source analysis to identify dependencies on other software

Introduction

1.2 Overview of the Porting Process

3. Selection of a migration method: rebuilding from sources or using a binary translator

Completing these steps yields the information necessary to write an effective migration plan.

The first step in evaluating an application for migration is to determine exactly what has to be migrated. This includes not only the application itself, but everything that the application requires in order to run properly. To begin evaluating your application, identify and locate the following items:

- Parts of the application
 - Source modules for the main program
 - Shareable images
 - Object modules
 - Libraries (object module, shareable image, text, or macro)
 - Data files and databases
 - Message files
 - Documentation
- Other software on which your application depends; for example,
 - Run-time libraries
 - HP Layered products
 - Third-party layered products
- Required operating environment
 - System characteristics
What sort of system is required to run and maintain your application; for example, how much memory is required, how much disk space, and so on?
 - Build procedures
This includes HP tools such as Code Management System (CMS) and Module Management System (MMS).
 - Testing Suite
You will need your tests to confirm that the migrated application runs correctly and to evaluate its performance.

Many of these items have already been migrated to HP OpenVMS I64; for example:

- HP Software bundled with the OpenVMS operating system
 - Run-time libraries
 - Other shareable libraries, such as those supplying callable utility routines and other application library routines
- HP layered products
 - Compilers and compiler RTLs
 - Database managers
 - Networking environment

1.2 Overview of the Porting Process

- Third-party products

Many third-party products now run on OpenVMS I64. To determine whether a particular application has been migrated, contact the application vendor.

You will be responsible for migrating your application and your development environment, including build procedures and test suites.

When you have completed the evaluation of your application, continue with a more detailed evaluation of each module and image, as described in Chapter 4.

Fundamental Differences to Consider

This chapter discusses the fundamental differences between the Alpha and I64 architectures.

2.1 Calling Standard

As with other components, the implementation of the OpenVMS Calling Standard on the Intel Itanium processor family sought to differ as little as possible from the OpenVMS VAX and Alpha conventions. The design methodology involved starting with Itanium conventions and making changes only where necessary. This helped to maintain compatibility with historical OpenVMS design while minimizing the cost and difficulty of porting applications and the operating system itself to the Itanium architecture.

The following sections contain a high-level description of differences between the *Itanium® Software Conventions and Runtime Architecture Guide* and this Calling Standard. For more detailed information, see the *HP OpenVMS Calling Standard*.

2.1.1 Changes to the Calling Standard

Data Model – OpenVMS on Alpha systems is deliberately ambiguous about the data model in use: many programs are compiled using what appears to be an ILP32 model, yet most of the system operates as though using either a P64 or LP64 model. The sign extension rules for integer parameters play a key role in making this more or less transparent. OpenVMS I64 preserves this characteristic, while the Itanium conventions define a pure LP64 data model.

Data Terminology – This specification uses the terms *word* and *quadword* to mean 2 bytes and 8 bytes, respectively, while the Itanium terminology uses these words to mean 4 bytes and 16 bytes respectively.

General Register Usage—General registers are used for integer arithmetic, some parts of VAX floating-point emulation, and other general-purpose computation. OpenVMS uses the same (default) conventions for these registers except for the following cases:

- R8 and R9 (only) are used for return values.
- R10 and R11 are used as scratch registers and not for return values.
- R25 is used for an AI (argument information) register.

Floating-Point Register Usage—Floating-point registers are used for floating-point computations, some parts of VAX floating-point emulation, and certain integer computations. OpenVMS uses the same (default) conventions for these registers except for the following cases:

- F8 and F9 (only) are used for return values.
- F10 through F15 are used as scratch registers and not for return values.

Fundamental Differences to Consider

2.1 Calling Standard

Parameter Passing—OpenVMS parameter passing is similar to the Itanium conventions, but with the following differences:

- Add an argument information register (for argument count and parameter type information).
- No argument is ever duplicated in both general and floating-point registers.
- For parameters that are passed in registers, the first parameter is passed in either the first general register slot (R32) or the first floating point register slot (F16), the second parameter in either the second general register (R33) or second floating register (F17) slot, and so on. Floating point parameters are not packed into the available floating point registers and at most eight parameters total are passed in registers.
- For 32-bit parameters passed in the general registers, the 32-bit value is sign-extended to the full 64-bit width of the parameter slot by replicating bit 31 (even for unsigned types).
- There is no even slot alignment for arguments larger than 64-bits.
- There is no special handling for HFA (homogeneous floating-point aggregates) in general, although some rules for complex types have a similar benefit.
- OpenVMS implements `__float128` pass-by value semantics using a reference mechanism.
- OpenVMS supports only little-endian representations.
- OpenVMS supports three additional VAX floating-point types for backward compatibility: `F_floating` (32 bits), `D_floating` (64 bits), and `G_floating` (64 bits). Values of these types are passed using the general registers.

Return Values – Return values up to at most 16 bytes in size may be returned in registers; larger return values are returned using a *hidden parameter* method using the first or second parameter slot.

2.1.2 Extensions to the Calling Standard

This section describes additions to or extensions of the Calling Standard.

Floating-Point Data Types – The calling standard for OpenVMS Itanium® includes support for the VAX `F_floating` (32-bit), `D_floating` (64-bit) and `G_floating` (64-bit) data types found on VAX and Alpha systems; it omits support for the Itanium 80-bit double-extended floating-point type.

VAX Compatible Record Layout—The OpenVMS standard adds a user optional VAX compatible record layout.

Linkage Options—OpenVMS allows additional flexibility and user control in the use of the static general registers as inputs, outputs, global registers and whether used at all.

Memory Stack Overflow Checking—OpenVMS defines how memory stack overflow checking should be performed.

Function Descriptors—OpenVMS defines extended forms of function descriptors to support additional functionality for bound procedure values and translated image support.

Unwind Information – OpenVMS adds an operating system-specific data area to the Itanium unwind information block. The presence of an operating system-specific data area is indicated by a flag in the unwind information header.

Handler Invocation – OpenVMS does not invoke a handler while control is in either a prologue or epilogue region of a routine. This difference in behavior is indicated by a flag in the unwind information header.

Translated Images – OpenVMS adds support (signature information and special ABIs) for calls between native and translated VAX or Alpha images.

2.2 Exception Handling

TBS

2.3 Floating-Point Arithmetic

This section discusses the differences in floating-point arithmetic on OpenVMS VAX, OpenVMS Alpha, and OpenVMS I64 systems.

2.3.1 Overview

The Alpha architecture supports both IEEE and VAX floating-point formats in hardware. OpenVMS compilers generate code using the VAX formats by default, with options on Alpha to use IEEE formats. The Itanium architecture implements floating-point arithmetic in hardware using the IEEE floating-point formats, including IEEE single and IEEE double.

If an application was originally written for OpenVMS VAX or OpenVMS Alpha using the default floating-point formats, it can be ported to OpenVMS I64 in one of two ways: continue to use VAX floating-point formats utilizing the conversion features of the HP compilers or convert the application to use IEEE floating-point formats. VAX floating-point formats can be used in those situations where access to previously generated binary floating-point data is required. HP compilers will generate the code necessary to convert the data between VAX and IEEE formats. For details about the conversion process, see the Floating Point White Paper at http://www.hp.com/products1/evolution/alpha_retaintrust/download/i64-floating-pt-wp.pdf. IEEE floating-point format should be used in those situations where VAX floating-point formats are not required. The use of IEEE floating-point formats will result in more efficient code.

2.3.2 Impact on OpenVMS Applications

VAX floating-point formats are supported on the Itanium architecture by converting them to IEEE single and IEEE double floating types. By default, this is a transparent process that will not impact most applications. HP is providing compilers for C, C++, Fortran, BASIC, PASCAL, and COBOL, all with the same floating-point options. Application developers need only recompile their applications using the appropriate compiler. Please see the individual compilers' documentation for detailed descriptions of floating-point options.

Because IEEE floating-point format will be the default, unless an application explicitly specifies VAX floating-point format options, a simple rebuild for OpenVMS I64 will use native IEEE formats directly. For programs that do not directly depend on the VAX formats for correct operation, this is the most desirable way to build for OpenVMS I64.

Fundamental Differences to Consider

2.4 Object File Format

2.4 Object File Format

Hewlett-Packard has determined that the most efficient way to utilize the compiler technologies developed by Intel is to adopt two industry standards, both of which are used by the Intel compiler:

- ELF (executable and linkable format) describes the object file and executable file formats.
- DWARF (debugging and traceback information format) describes the way debugging and traceback information is stored in the object and executable files.

One clear advantage of this decision is that development tools will be ported more easily to OpenVMS in the future. All of the compilers, development tools, and utilities provided with the OpenVMS operating system on the Itanium architecture will utilize and understand these new formats. Application developers will not need to be concerned with these formats unless their applications have specific knowledge of them.

In a typical application scenario in which an application is compiled, linked, debugged, and deployed, the details of the object file format, the executable image file format, and the debugging and traceback information are of no concern to the developer. However, software engineers who develop compilers, debuggers, analysis tools, or other utilities that are dependent on these file formats will need to understand how they are being implemented on OpenVMS.

Assessing Needs for Porting Your Application

Typically, porting an application to a new platform requires the following tasks:

1. Assessing porting needs
2. Identifying architecture dependencies and non-standard coding practices in the application to be ported, and rewriting code accordingly
3. Compiling, linking, and running the application
4. Performing testing before and after porting
5. Recompiling the code and repeating the above steps, as necessary
6. Shipping the revised application
7. Operating and installing processes or procedures
8. Supporting and maintaining the application

Although you complete each task sequentially, you might have to repeat some steps if problems arise.

This chapter addresses the first task, assessing porting needs. Subsequent chapters discuss the remaining tasks. This chapter describes the responsibilities and issues that developers should consider in planning to port an application. In particular, it sets the context for porting applications from an OpenVMS Alpha to an OpenVMS I64 environment and provides the information necessary to help developers evaluate needs, plan, and begin the software porting.

3.1 Overview

Most applications running on OpenVMS Alpha today can be ported easily to OpenVMS I64 with few changes. The Alpha and I64 variants of OpenVMS are produced from a single source code base, enabling developers to incorporate features (that are independent of the hardware) into both versions without requiring multiple changes to the source code. This minimizes the time required to perform qualification testing and helps to ensure the availability of critical applications on both OpenVMS platforms.

HP intends to maintain a strict policy of ensuring and maintaining forward source compatibility so that "well-behaved" applications that run on recommended versions of OpenVMS Alpha will also run successfully on OpenVMS I64 systems. For the most part, if an application takes advantage of published system services and library interfaces, the application should be portable (as is) to the latest version of OpenVMS I64. However, some published system and library interfaces available on OpenVMS Alpha will not be available or will behave differently on OpenVMS I64. In addition, the Linker utility does not support all qualifiers supported on OpenVMS Alpha. For more information about these exceptions, see Chapter 4 and Chapter 5.

Assessing Needs for Porting Your Application

3.1 Overview

OpenVMS I64 has the same look and feel that OpenVMS customers are familiar with. Minor changes were needed to accommodate the new architecture, but the basic structure and capabilities of OpenVMS are the same.

3.2 Evaluating the Application

The first step to porting an application is to evaluate and identify the steps necessary to port the application to OpenVMS I64. The evaluation process culminates in a porting plan that answers the following questions:

- How do you port the application?
- How much time, effort, and cost will porting require?
- Do you require support from HP services?

The evaluation process has the following four stages:

1. General inventory of the application itself
2. Source analysis to identify dependencies on other software
3. Selection of a porting method
4. Analysis of operational tasks

Completing these steps yields the information necessary to write an effective porting plan. Sections 3.2.1 through 3.2.4 suggest the steps for an initial, rudimentary evaluation, after which you can perform a more detailed evaluation of each module, image, and all dependencies. Section 3.5 includes a detailed checklist with commentary to help guide you through planning, evaluating, and porting your application successfully. Appendix A provides a more usable version of the checklist that omits the commentary.

After you complete the evaluation steps suggested in the sections to follow, compare your inventory with the supported product, procedures, and functionality of the target platform. Research any deviations for schedule mismatches, missing functionality, procedural conflicts, and supportability. To complete your porting plan, include all costs and time impacts.

3.2.1 Selecting the Porting Method

When you complete the evaluation process described in the sections to follow, you will have to port your application and your development environment. In addition to the source modules, you might have to port other components such as build procedures, test suites, and in some cases, the application data structures. You must decide on the best method for porting each part of the application. Usually the issue is whether to rebuild your application from sources or to use a binary translator. To make this decision, you will need to know which methods are possible for each part of the application, and how much work will be required for each method. The evaluation process consists of a series of questions and some tasks to perform to help answer those questions, as shown in Figure 3–1.

The majority of applications can be ported by recompiling and relinking them. If your application runs only in user mode and is written in a standard high-level language, it is most likely in this category.

Note

The current version of OpenVMS I64 (Version 8.1) does not include the binary translator (BT).

3.2.2 Taking Inventory of What Is to Be Ported

The first step in evaluating an application for porting is to take inventory of exactly what has to be ported. This includes not only the application itself, but everything that the application requires to run properly. To begin evaluating your application, identify and locate the following items:

- Source modules for the main program
- Shareable images
- Object modules
- Libraries (object module, shareable image, text, or macro)
- Data files and databases
- Message files
- Scripts and procedures (build files)
- Application documentation (if any)

Consider the differences between OpenVMS Alpha and OpenVMS I64 described in Chapter 2 and the consequent changes that are necessary for application components prior to porting. In addition, consider the processor-related development issues, as described in Chapter 7.

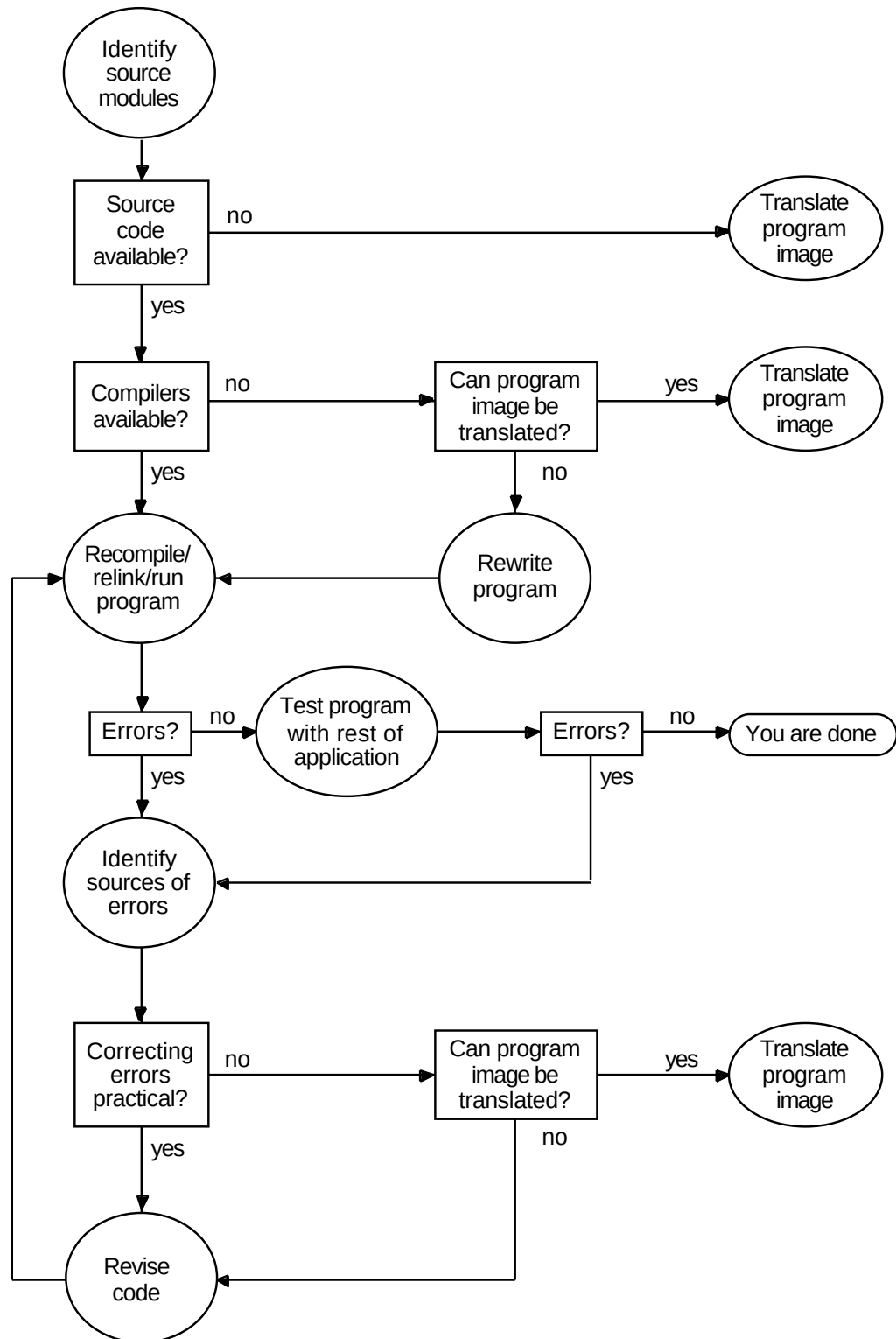
Evaluate the time and costs required for these changes. Any code that depends specifically on the OpenVMS Alpha architecture will have to be changed. For example, an application might require changes if it includes code that:

- Deals with specific machine instructions or that makes assumptions about the number of registers or the functions of specific registers
- Relies on particular elements of the OpenVMS Alpha calling standard (for differences between OpenVMS Alpha and OpenVMS I64 calling standards, see Section 2.1.1)
- Relies on unpublished system services and library interfaces, or certain system and library interfaces that are not available or will behave differently on OpenVMS I64 (for more information on the availability and divergent behavior of certain system and library interfaces, see Chapter 5 as well as Chapter 4)
- Relies on the OpenVMS Alpha object file format, the executable image file format, or the debug symbol table format
- Is conditionalized, or includes logic that assumes it is running on either an OpenVMS VAX or an OpenVMS Alpha system might not be able to handle a third platform

Assessing Needs for Porting Your Application

3.2 Evaluating the Application

Figure 3–1 Porting an Application



ZK-4990A-GE

Assessing Needs for Porting Your Application

3.2 Evaluating the Application

- Depends on OpenVMS Alpha or VAX internal data structures; for example:
 - Floating point formats that are not IEEE standard (IEEE floating point formats are the default for OpenVMS I64 and should be used where VAX floating-point formats are not required; for more information, see Section 3.2.4 and Section 4.8.4)
 - The mechanism array data structure on OpenVMS I64 differs significantly from that on OpenVMS Alpha (see Section 4.8.3.3)

Note

HP recommends that you align your data naturally to achieve optimal performance for data referencing. On both OpenVMS Alpha and OpenVMS I64 systems, referencing data that is unaligned degrades performance significantly. In addition, unaligned shared data can cause a program to execute incorrectly. You must align shared data naturally. Shared data might be between threads of a single process, between a process and ASTs, or between several processes in a global section. For more information about data alignment, see Section 4.8.6 in this book and the *OpenVMS Programming Concepts Manual*.

- References terminal drivers that do not use the call interface; for example, those using Jump to Subroutine (JSB) macros need to be revised to use the call interface
- Is user-written threading-like (for example, code that performs direct stack switching, implements a co-routine or tasking model, or depends on the nature of the procedure call stack frames; for more details, see Section 4.8.5)
- Incorporates non-standard or undocumented code practices or interfaces

In addition any source code that does not have a supported compiler on OpenVMS I64 will have to be rewritten or translated (for a list of supported compilers, see Chapter 6). Note also that certain language capabilities of OpenVMS Alpha are not supported by OpenVMS I64; for more information, see *HP OpenVMS Industry Standard 64 for Integrity Servers Version 8.1 New Features and Release Notes*. In addition, applications that run in privileged mode might require changes.

For more details about changes required in your source modules prior to porting them, see Chapter 4.

3.2.3 Assessing Dependencies

The next step is to assess the software and environment that your application depends on.

3.2.3.1 Software Dependencies

Software dependencies might include such components as:

- Run-time libraries
- HP layered products
- Third-party layered products
- Tools

Assessing Needs for Porting Your Application

3.2 Evaluating the Application

Many of these items have already been migrated to OpenVMS I64, including the following:

- HP software bundled with the OpenVMS operating system, including:
 - Run-time libraries
 - Other shareable libraries, such as those supplying callable utility routines and other application library routines
- HP layered products, including:
 - Compilers and compiler RTLs
 - Database managers
 - Networking environment

- Third-party products

Many third-party products now run on OpenVMS I64. To determine whether a particular application has been migrated, contact the application vendor.

Important

The availability of third-party software can be the biggest obstacle for a porting project. You must determine the availability and cost of third-party software for the target operating system. This might include build environment tools, third-party libraries, and automated testing tools. For information about tools that are freely available on OpenVMS, see Chapter 5.

3.2.3.2 Development Environment

Consider the development environment upon which your application depends, including the operating system configuration, hardware, development utilities, build procedures (including HP tools such as Code Management System (CMS) and Module Management System (MMS)), and so forth.

3.2.3.3 Operating Environment

Consider the following issues regarding the operating environment:

- System characteristics

What sort of system is required to run and maintain your application; for example, how much memory is required, how much disk space, and so on?
- Testing suite

Your tests should confirm that the ported application runs correctly, and they should evaluate its performance. Regression tests aid in testing software on new versions of the operating system and platform. The regression tests can be any one or combination of the following:

 - A software program written by your development team to exercise all possible code paths and functions
 - A series of DCL command procedures
 - Interactive testing conducted manually (instead of automatically by software)
 - HP's Digital Test Manager or an equivalent Open Source tool

Assessing Needs for Porting Your Application

3.2 Evaluating the Application

Regression testing is crucial to the development of any new product and especially to the porting of an application to a new system.

3.2.4 Operational Tasks

Evaluate the responsibilities and needs for operational tasks required to port and maintain the applications, such as:

- Installation requirements
- Compilation requirements
 - Is the compiler that you use for your application also supported on the current version of OpenVMS I64?
 - Prior to porting your application, HP recommends that you compile your application on an OpenVMS Alpha system using the latest version of the OpenVMS Alpha compiler. In this way, you can uncover problems that you might encounter when you compile the ported application on OpenVMS I64. Some newer versions of compilers apply a stricter interpretation of existing compiler standards or enforce newer, stricter standards. For more information about compilers supported on the current version of OpenVMS I64, see Chapter 6.
 - After testing with a compile on the OpenVMS Alpha system and resolving any problems, you need to port the application to your OpenVMS I64 system, recompile, relink, and requalify. Plan for these activities accordingly.
 - Can you switch to IEEE floating point data types instead of VAX floating point data types? In general, applications can still use VAX floating point format on OpenVMS I64. Use VAX floating point formats where access to previously generated binary floating point data is required. VAX floating point is achieved on OpenVMS I64 systems in software. HP compilers will automatically generate the code necessary to convert VAX floating point values to the IEEE floating point format and perform the mathematical operation; however, they then convert the value back to VAX floating point. This additional conversion adds execution overhead to the application. For this reason, consider switching your application to use IEEE floating point format.

Applications on OpenVMS Alpha can be switched to use IEEE floating point with no performance penalty because IEEE floating point is supported by the Alpha hardware. This will make migration of your application to OpenVMS I64 easier. IEEE floating point is the only floating point format supported by the Itanium hardware. In addition, your application will perform better.

You can test an application's behavior with IEEE floating-point values by compiling it on an OpenVMS Alpha system with an IEEE qualifier (specifying the compiler option `/FLOAT=IEEE`). If that produces acceptable results, you should simply build the application on the OpenVMS I64 system (and Alpha, if you wish) using the same qualifier.

For more information, see Section 4.8.4 and the "OpenVMS floating-point arithmetic on the Intel Itanium Architecture" white paper. You can obtain the white paper from the following location:

http://www.hp.com/products1/evolution/alpha_retaintrust/openvms/resources.html

- Backup and restore functionality

Assessing Needs for Porting Your Application

3.2 Evaluating the Application

- Operator interface look and feel
- System administration

3.3 HP Porting Resources

No one organization has the expertise in all phases of a porting process. Therefore, HP has gathered a set of resources that can assist you through the process. These resources can provide everything from porting guides to teams of personnel with equipment that handles the entire porting task. For more information, consult your local HP representative and the Alpha Retain Trust services web site at:

http://www.hp.com/products1/evolution/alpha_retaintrust/services.html

A sampling of other resources available include the following:

- Developer and Solution Partner Program (DSPP)

<http://www.hp.com/go/dspp>

- Test Drive Program

<http://www.testdrive.hp.com/>

- HP Services

<http://www.hp.com/go/services/>

As HP's Itanium-architecture strategy continues to develop, Sales Resources and Global Services will be assessing the impact to the long-term plans of HP customers and ISVs. HP will create and document template and custom service offerings to address generic and unique situations of each customer. HP will tailor each service offering to support your needs by making available the tools and resources needed to assist with crucial changes during the transition.

3.4 Additional Considerations

HP recognizes that not all layered software and middleware supplier products will port to OpenVMS I64 at first release. HP is committed to assist software vendors with information, hardware, support, and tools that will facilitate this process. Each vendor should monitor the progress of the supplier of their prerequisite software to ensure that the required components are in place for their development schedules.

HP intends to implement a tool suite on the existing OpenVMS Alpha platform that will make porting to the new OpenVMS I64 platform transparent for most developers in terms of the code base, commands, and process. Nothing will replace the testing that accompany any quality product release. Rather, this tool suite will reduce the time required for product porting.

3.5 Porting Checklist

Use the following checklist to guide you through the planning and porting process. Comments in brackets following a question or step intend to help clarify the purpose of that item. Appendix A includes a more concise version of this checklist without the commentary.

Assessing Needs for Porting Your Application

3.5 Porting Checklist

General Information

1. What does your application do (brief description)?

Development History and Plans

2. a. Does the application currently run on other operating systems or hardware architectures? ☐ YES ☐ NO
- b. If yes, does the application currently run on the latest OpenVMS Alpha system? ☐ YES ☐ NO

[If your application already runs on multiple platforms, it is likely to be platform-independent and therefore easier to port to OpenVMS I64. (However, if the application code is conditionalized per platform, then minor modifications might be required. For more information, see Section 4.8.1.) If you answer YES to b, your application will be easier to port to OpenVMS I64. If your application is running on OpenVMS VAX, you must first port it to OpenVMS Alpha (for more information, see the *Migrating an Application from OpenVMS VAX to OpenVMS Alpha* manual).]

3. When was the last time the application environment was completely recompiled and rebuilt from source? _____

- a. Is the application rebuilt regularly? ☐ YES ☐ NO
- b. How frequently? _____

[Applications that are not built frequently might require additional work before being ported. For example, changes in the development environment might cause incompatibilities. Prior to porting your applications, confirm that they can be built on the latest version of OpenVMS Alpha.]

4. Is the application actively maintained by developers who know it well? ☐ YES ☐ NO

List developer names and contact information:

Developer	Contact
_____	_____
_____	_____
_____	_____

5. a. How is a new build of the application tested or verified for proper operation?

- b. Do you have performance characterization tools to assist with optimization? ☐ YES ☐ NO

- c. If yes, list the tools and version numbers used:

Assessing Needs for Porting Your Application

3.5 Porting Checklist

Tool	Version

d. Which source-code configuration management tools are used?
List the tools and their version numbers:

Tool	Version

6. Do you have a development and test environment separate from your production systems? ☐ YES ☐ NO

[OpenVMS I64 Version 8.1 is for evaluation purposes only; it is not intended for use in production environments.]

7. What procedures are in place for rolling in a new version of the application into production?

8. What are your plans for the application after porting it to OpenVMS I64?

- | | | |
|--|------------------------------|-----------------------------|
| a. No further development | ☐ YES | ☐ NO |
| b. Maintenance releases only | ☐ YES | ☐ NO |
| c. Additional or changed functionality | ☐ YES | ☐ NO |
| d. Maintain separate OpenVMS I64 and OpenVMS Alpha sources | ☐ YES | ☐ NO |

[If you answer YES to a, you may wish to consider translating the application. A YES response to b or c should give you reason to evaluate the benefits of recompiling and relinking your application, although translation is still possible. If you intend to maintain separate OpenVMS I64 and OpenVMS Alpha sources, as indicated by a YES to d, you may need to consider interoperability and consistency issues, especially if the different versions of the application can access the same database.]

Composition of the Application

Consider the size and components of your application that need to be ported.

9. How large is your application?

How many modules?

How many lines, disk blocks, or megabytes of code?

How much disk space is required?

[This will help you "size" the effort and the resources required for porting. Most helpful is knowing the number of disk blocks of code.]

Assessing Needs for Porting Your Application

3.5 Porting Checklist

10. a. Do you have access to all source files that make up your application? ☐ YES ☐ NO
- b. If you are considering using HP Services, will it be possible to give HP access to these source files and build procedures? ☐ YES ☐ NO

[If you answer NO to a, translation may be your only porting option for the files with missing sources. Translate user-mode OpenVMS Alpha images to OpenVMS I64 images.]

Note

The current version of OpenVMS I64 (Version 8.1) does not include the binary translator (BT).

Note that if you have OpenVMS VAX images, first translate them to OpenVMS Alpha images. You cannot translate them directly to OpenVMS I64 images.

A YES answer to b allows you to take advantage of a greater range of HP porting services.]

11. a. List the languages used to write the application. (If multiple languages are used, give the percentages of each.)

Language	Percentage
_____	_____
_____	_____
_____	_____

- b. Do you use Macro-64, PL/I, ADA 83, or Fortran 77 on OpenVMS Alpha? ☐ YES ☐ NO

[If you use Macro-64 on OpenVMS Alpha, you must rewrite the code because no Macro-64 compiler exists for OpenVMS I64. If possible, HP recommends rewriting the code into a high-level language using documented system interfaces. For machine-specific code that must be written in assembly language, you must recode the Macro-64 code into OpenVMS I64 assembly code.]

Similarly, PL/I, Fortran 77, and ADA 83 are not supported on OpenVMS I64. If your application has code written in PL/I, HP recommends rewriting it in another language such as C or C++. Update code written in Ada 83 to Ada 95, and code written in Fortran 77 to Fortran 90. Analyze your code to determine if the newer compiler will require changes to the code. If the compilers are not yet available, you must translate or rewrite in a different language.

In general, if the compilers are not available on OpenVMS I64, you must translate or rewrite code in a different language.

For information on availability of compilers and translators, see Chapters 5 and 6.]

12. Is there application documentation? ☐ YES ☐ NO

[If you answer yes, note that if any changes were required to the application, the documentation might have to be updated accordingly.]

Assessing Needs for Porting Your Application

3.5 Porting Checklist

External Dependencies

Consider the system configuration, environment, and software required for developing, testing, and running your application.

13. What is the system configuration (CPUs, memory, disks) required to set up a development environment for the application?

[This will help you plan for the resources needed for porting.]

14. What is the system configuration (CPUs, memory, disks) required to set up a typical user environment for the application, including installation verification procedures, regression tests, benchmarks, or workloads?

[This will help you determine whether your entire environment is available on OpenVMS I64.]

15. Does the application rely on any special hardware? ☐ YES ☐ NO

[This will help you determine whether the hardware is available on OpenVMS I64, and whether the application includes hardware-specific code.]

16. What version of OpenVMS Alpha does your application currently run on? _____

Does the application run on OpenVMS Alpha Version 7.3-2 or 7.3-1? ☐ YES ☐ NO

[HP recommends having your application running on the latest version of OpenVMS Alpha before porting it to OpenVMS I64.]

17. Does the application require layered and third-party products to run?

a. From HP: (other than compiler RTLs) ☐ YES ☐ NO

List the HP layered products and versions:

HP Product	Version
_____	_____
_____	_____
_____	_____

b. From third parties: ☐ YES ☐ NO

List the third-party products and versions:

Third-Party Product	Version
_____	_____
_____	_____

Assessing Needs for Porting Your Application

3.5 Porting Checklist

[If you answer YES to a and are uncertain whether the HP layered products are yet available for OpenVMS I64, check with an HP support representative. For information on HP's commitment to support porting of software, see Section 3.4. If you answer YES to b, check with your third-party product supplier.]

18. a. Do you have regression tests for the application? ☐ YES ☐ NO
b. If yes, do they require any particular software product such as HP's Digital Test Manager? ☐ YES ☐ NO

[If you answer YES to a, you should consider porting those regression tests. If you answer YES to b, HP's Digital Test Manager (part of the DECset product set) is available with this release of OpenVMS. If you require other tools for testing purposes, contact your HP support representative; see also Chapter 5. HP recommends a disciplined testing process to check regressions in product development.]

Dependencies on the OpenVMS Alpha Architecture

Consider differences between OpenVMS Alpha and OpenVMS I64, and between the OpenVMS Alpha and OpenVMS I64 processor families. The following questions will help you account for the most significant differences. Consider the changes that might be necessary to application components because of these differences. Any user-written code or assembly code that depends specifically on the OpenVMS Alpha architecture must be rewritten.

19. Does the application use OpenVMS VAX floating data types? ☐ YES ☐ NO
[If you answer YES, note that the default for OpenVMS I64 compilers is IEEE floating data types. OpenVMS I64 compilers provide for VAX floating support by automatically converting from OpenVMS VAX to IEEE floating data types. Slight precision differences might result. In addition, run-time performance will be somewhat slower (compared to the direct use of IEEE floating data types without conversion), depending on the extent to which the application is dominated by floating point calculations. For more information about floating point data types, see Section 4.8.4.]
20. Is the data naturally aligned? ☐ YES ☐ NO
[If your data is not naturally aligned, HP recommends that you align your data naturally to achieve optimal performance for data referencing. Unaligned data can degrade data referencing performance significantly. In certain instances, you must align data naturally. For more information about data structures and alignment, see Section 4.8.6 and *OpenVMS Programming Concepts Manual*.]
21. Does the application call OpenVMS system services that:
a. Modify the working set? ☐ YES ☐ NO
b. Change program control flow using SYS\$GOTO_UNWIND? ☐ YES ☐ NO

Assessing Needs for Porting Your Application

3.5 Porting Checklist

[If you answer YES to a, make sure that calls to the SYS\$LKWSET and SYS\$LKWSET_64 system services specify the required input parameters correctly.

If you answer YES to b, change any calls to SYS\$GOTO_UNWIND to SYS\$GOTO_UNWIND_64. OpenVMS I64 does not support SYS\$GOTO_UNWIND.

For more information, see Chapter 4.]

22. a. Does the application use multiple, cooperating processes? ☐ YES ☐ NO

If so:

- b. How many processes? _____

- c. What interprocess communication method is used? _____

☐ \$CRMPSC ☐ Mailboxes ☐ SCS ☐ Other
☐ DLM ☐ SHM, IPC ☐ SMG\$ ☐ STR\$

- d. If you use global sections (\$CRMPSC) to share data with other processes, how is data access synchronized? _____

[This will help you determine whether you will need to use explicit synchronization, and the level of effort required to guarantee synchronization among the parts of your application. Use of a high-level synchronization method generally allows you to port an application most easily.]

23. Does the application currently run in a multiprocessor (SMP) environment? ☐ YES ☐ NO

[If you answer YES, it is likely that your application already uses adequate interprocess synchronization methods.]

24. Does the application use AST (asynchronous system trap) mechanisms? ☐ YES ☐ NO

[If you answer YES, you should determine whether the AST and main process share access to data in process space. If so, you may need to explicitly synchronize such accesses.]

25. a. Does the application contain condition handlers? ☐ YES ☐ NO
b. Do any handlers modify data in the mechanism array? ☐ YES ☐ NO

[The OpenVMS Alpha and I64 mechanism array formats differ significantly. Be sure to modify any applications that modify register values in the mechanism array.]

26. a. Does the application run in privileged mode or link against SYS\$BASE_IMAGE? ☐ YES ☐ NO

If so, why? _____

- b. Does the application depend on OpenVMS internal data structures or interfaces? ☐ YES ☐ NO

Assessing Needs for Porting Your Application

3.5 Porting Checklist

[Applications that link against the OpenVMS executive or run in privileged mode might require additional porting work. Undocumented interfaces in SYS\$BASE_IMAGE might have changed on OpenVMS I64.]

Applications that depend on OpenVMS internal data structure definitions (as defined in SYS\$LIBRARY:LIB.INCLUDE, SYS\$LIBRARY:LIB.L32, SYS\$LIBRARY:LIB.MLB, SYS\$LIBRARY:LIB.R64, SYS\$LIBRARY:LIB.REQ, and SYS\$LIBRARY:SYS\$LIB_C.TLB) might also require additional porting work, as some internal data structures have changed on OpenVMS I64.]

27. Does the application use connect-to-interrupt mechanisms? ☐ YES ☐ NO
If yes, with what functionality? _____
[Connect-to-interrupt is not supported on OpenVMS I64 systems. Contact an HP representative if you need this feature.]
28. Does the application create or modify machine instructions? ☐ YES ☐ NO
[Guaranteeing correct execution of instructions written to the instruction stream requires great care on OpenVMS I64. Any code dealing with specific machine instructions must be rewritten.]
29. Does the application include any other user-written code or assembler code that depends specifically on the OpenVMS Alpha architecture? ☐ YES ☐ NO
[If you answer YES, rewrite such code. Optimally, rewrite it so that it is independent of architecture.]
30. Does the application include or depend on any conditionalized statements in source code or build files, or any code that includes logic that assumes it is running on either a VAX or an OpenVMS Alpha system? ☐ YES ☐ NO
[If you answer YES, you might have to modify the conditional directives slightly. For example, change "IF ALPHA" to "IF ALPHA OR IPF", or change "IF NOT ALPHA" to "IF VAX". For more information, see Section 4.8.1.]
31. What parts of the application are most sensitive to performance? I/O, floating point, memory, realtime (that is, interrupt latency, and so on).

[This will help you determine how to prioritize work on the various parts of your application and allow HP to plan performance enhancements that are most meaningful to customers.]
32. Does the application use any OpenVMS Alpha calling standard routines? ☐ YES ☐ NO

Assessing Needs for Porting Your Application

3.5 Porting Checklist

[If you answer YES, note that applications that use the calling standard data structure named LIBICB, or that make use of the calling standard routines such as LIB\$GET_CURRENT_INVO_CONTEXT, LIB\$GET_PREVIOUS_INVO_CONTEXT, LIB\$PUT_INVO_CONTEXT, and LIB\$GET_INVO_HANDLE, will have to be modified. These routines have been renamed in OpenVMS I64 due to changes in the LIBICB data structure. In addition, the calling standard has changed. Applications with direct knowledge of the OpenVMS Alpha calling standard must be modified for OpenVMS I64. For more information on the differences between OpenVMS Alpha and I64 calling standards, see Section 2.1.1. For more information on OpenVMS Alpha calling standards, see *HP OpenVMS Calling Standard*.]

33. Does the application use non-standard routine linkage declarations? ☐ YES ☐ NO

[If you answer YES, note that by definition applications with non-standard linkages know the OpenVMS Alpha calling standard and might need modification to conform to the OpenVMS I64 calling standard.]

34. a. Does the application depend on the format or content of an object file? ☐ YES ☐ NO
b. Does the application depend on the format or content of an executable image? ☐ YES ☐ NO
c. Does the application depend on the debug symbol table content of an image? ☐ YES ☐ NO

[If you answer YES to a, b, or c, note that applications that depend on any of these image and object file data structures must be modified. The industry-standard object and image file layout called ELF has been adopted by OpenVMS I64. The industry-standard DWARF debug symbol table format has also been adopted. Applications with dependencies on OpenVMS Alpha's object, image or debug symbol table formats must be modified. For more information, see Section 4.8.3.4.]

35. Does the application use the C asm statement? ☐ YES ☐ NO

[If you answer YES, note that applications that use the C asm statement have Alpha instruction dependencies and must be modified. The HP C compiler for OpenVMS I64 has added many new builtin functions that minimize the need for using asm statements.]

36. Does the application use BLISS register BUILTINS? ☐ YES ☐ NO

[If you answer YES, note that applications that use the BLISS BUILTIN statement to declare register builtins must be modified.]

Porting Your Application

Once the preliminary evaluation and planning stages are completed, these are the main porting tasks.

37. Have you upgraded your OpenVMS Alpha system to the latest version currently available? ☐ YES ☐ NO

Assessing Needs for Porting Your Application

3.5 Porting Checklist

[If you answer NO, first upgrade your OpenVMS Alpha system before beginning to port the application.]

38. a. Have you test compiled your application on the latest version of OpenVMS Alpha, using the most recent version of the compiler? ☐ YES ☐ NO

- b. Have you corrected any problems detected during the compile? ☐ YES ☐ NO

[If you answer NO to a, compile your application on the latest OpenVMS Alpha system, using the latest compiler available. If you answer YES to a but have rewritten some code since, compile again. After compiling, correct any problems detected. When you have finished this phase, and when you have responded to all other considerations detailed thus far in this checklist and in Chapters 4 through 7, your application is ready for porting to OpenVMS I64.]

39. Copy the application source modules and all related code to the OpenVMS I64 system. Have you copied the modules and related code to OpenVMS I64? ☐ YES ☐ NO

40. Compile, link, and run the application on the OpenVMS I64 system.
Application is compiled, linked, and running? ☐ YES ☐ NO
[For details, refer to Chapter 5.]

41. Perform unit testing on the OpenVMS I64 system.
Have you completed the unit testing? ☐ YES ☐ NO
[For details, refer to Chapter 5.]

42. Perform system testing on the OpenVMS I64 system.
Have you completed the system testing? ☐ YES ☐ NO
[For details, refer to Chapter 5.]

43. Perform regression testing on the OpenVMS I64 system.
Have you completed the regression testing? ☐ YES ☐ NO
[For details, refer to Chapter 5.]

44. Did testing succeed? ☐ YES ☐ NO

[If you answer no, resolve the problems, then recompile, relink, and retest the application. For details, refer to Chapter 5.]

Migrating Source Modules

This chapter describes the steps common to porting all applications. It then describes certain types of code and coding practices that may need to be changed before you can recompile your application.

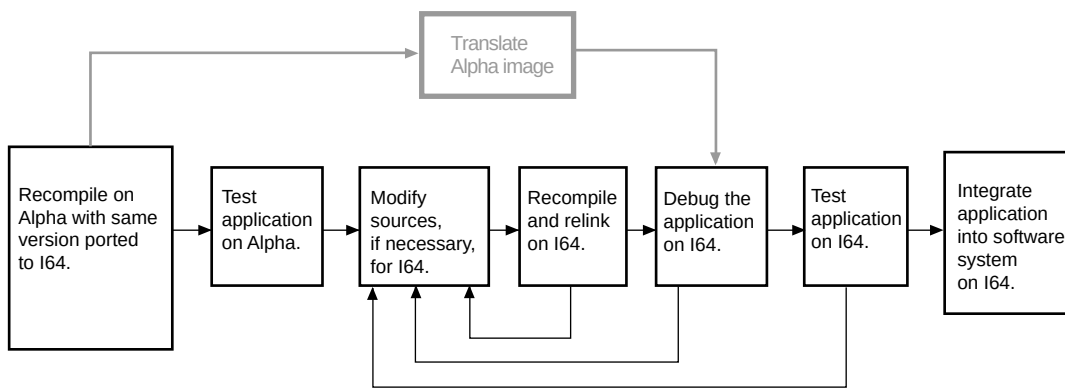
If you have thoroughly analyzed your code and planned the migration process, the final stage of migrating source modules should be fairly straightforward. You may be able to recompile many programs with no change. Programs that do not recompile directly will frequently need only straightforward changes to get them up on an I64 system. Certain types of code will require changes.

Migrating your application involves the following steps:

1. Setting up the migration environment.
2. Compiling your application on an Alpha system with the latest versions of the Alpha compilers.
3. Testing the application on an Alpha system to establish baselines for evaluating the migration.
4. Recompiling and relinking an application on an I64 system.
5. Debugging the migrated application.
6. Testing the migrated application.
7. Integrating the migrated application into a software system.

These steps, except for setting up the migration environment, are illustrated in Figure 4–1. Note that the binary translator for translating Alpha images is not yet available.

Figure 4–1 Migrating an Application from Alpha to I64



VM-1121A-AI

Migrating Source Modules

If your source files possess any of the following characteristics, you may need to change them:

- Use VAX MACRO-32 or Alpha MACRO-64 assembly language
- Use conditions to produce the same application for different platforms from a common code base
- Rely on the Alpha architecture, such as the Alpha object code format
- Use floating point data types
- Use threads
- Contain unaligned data
- Rely explicitly on the OpenVMS Alpha calling standard
- Use privileged interfaces or operate at inner access modes

For more information, see Section 4.8.

4.1 Setting Up the Migration Environment

The native I64 environment is comparable to the development environment on Alpha systems. Most of the program development tools that you are accustomed to using on OpenVMS Alpha are available. Some modifications have been made to these tools to account for the differences in the two architectures and the calling standard. Most of these modifications are invisible to users.

An important element of the I64 migration process is the availability of support from HP Services, which can provide help in modifying, debugging, and testing your application. For more information, see Section 3.3.

4.1.1 Hardware

There are several issues to consider when planning what hardware you will need for your migration. To begin, consider what resources are required in your normal Alpha development environment:

- CPUs
- Disks
- Memory

In an I64 environment, the following configuration is recommended for this release:

- HP rx2600 system with 1 or 2 processors
- 1 to 24 GB of memory
- 1 to 4, 36-GB disks
- DVD/CD-ROM

For more information, refer to the *HP OpenVMS Industry Standard 64 for Integrity Servers Version 8.1 New Features and Release Notes*.

4.1.2 Software

To create an efficient migration environment, check the following elements:

- Migration tools
You need a compatible set of migration tools, including the following:
 - Compilers
 - RTLs
 - System libraries
 - Include files for C programs
- Logical names
Logical names must be consistently defined to point to Alpha and I64 versions of tools and files.
- Compile and link procedures
These procedures may need to be adjusted for new tools and the new environment.
- Test tools
You will need to port the Alpha test tools to OpenVMS I64, unless they are already ported. You will also need test tools that are designed to test for OpenVMS I64 characteristics, such as changes to the OpenVMS calling standard that are specific to OpenVMS I64.
- Tools for maintaining sources and building images, such as:
 - CMS (Code Management System)
 - MMS (Module Management System)

4.2 Compiling Applications on Alpha With Current Compiler Version

Before recompiling your code with a native I64 compiler, HP recommends that you first compile the code to run on Alpha with the latest version of the compiler. For example, if the application is written in Fortran, make sure that the application has been recompiled with Fortran Version 7.5 (Fortran 90) to run on Alpha. Fortran Version 7.5 is the version that was ported to I64.

Using the latest version of the compiler to recompile your application on OpenVMS Alpha might uncover problems that are attributable to the change in the compiler version only. For example, newer versions of compilers may enforce programming language standards that were previously ignored, exposing latent problems in your application code. Newer versions may also enforce changes to the standard that were not in effect when the earlier versions were created. Fixing such problems on OpenVMS Alpha simplifies porting that application to I64.

For a list of the native OpenVMS I64 compilers, see Section 5.1.

4.3 Testing Applications on Alpha for Baseline Information

The first step in testing is to establish baseline values for your application by running your test suite on the Alpha application. You can do this before or after you port your application to I64. You can then compare the results of these tests with the results of similar tests on an I64 system, as described in Section 4.6.

4.4 Recompiling and Relinking on an I64 System

In general, migrating your application involves repeated cycles of revising, compiling, linking, and debugging your code. During the process, you will resolve all syntax and logic errors noted by the development tools. Syntax errors are usually simple to fix; logic errors typically require significant modifications to your code.

Your compile and link commands will likely require some changes, such as new compiler and linker switches. For example, the new Linker qualifier `/BASE_ADDRESS` replaces the `CLUSTER=(base-address)` option on Alpha, which is illegal on I64. For more information about the new linker qualifiers, see Chapter 5. For more information about the compiler switches, see Chapter 6. If you are porting VAX MACRO code, refer to the *HP OpenVMS MACRO Compiler Porting and User's Guide*.

4.5 Debugging the Migrated Application

Once you have migrated your application to OpenVMS I64, you may have to debug it. This section describes the OpenVMS tools available for debugging your application.

For more information about these tools, see Chapter 5, Chapter 6, and the *HP OpenVMS Industry Standard 64 for Integrity Servers Version 8.1 New Features and Release Notes*.

4.5.1 Debugging

The OpenVMS I64 operating system provides the following debuggers:

- The OpenVMS Debugger
The OpenVMS Debugger is a symbolic debugger; that is, the debugger allows you to refer to program locations by the symbols you used for them in your program—the names of variables, routines, labels, and so on. You do not need to specify memory addresses or machine registers when referring to program locations, although you can if you wish.
- The XDelta Debugger
The XDelta Debugger is an address location debugger; that is, the debugger requires you to refer to program locations by address location. This debugger is primarily used to debug programs that run in privileged processor mode or at an elevated interrupt level. The related Delta Debugger is not yet available.

The System-Code Debugger is a symbolic debugger that allows you to debug nonpageable code and device drivers running at any IPL. This debugger will be available on I64 in a future release.

4.5.2 Analyzing System Crashes

OpenVMS provides two tools for analyzing system crashes: the System Dump Analyzer and the Crash Log Utility Extractor.

4.5.2.1 System Dump Analyzer

The System Dump Analyzer (SDA) utility on OpenVMS I64 systems is almost identical to the utility provided on OpenVMS Alpha systems. Many commands, qualifiers, and displays are identical, including several for accessing functions of the Crash Log Utility Extractor (CLUE) utility. Some displays have been adapted to show information specific to OpenVMS I64 systems.

To use SDA on an Alpha computer, you must first familiarize yourself with the OpenVMS calling standard for Alpha systems. Similarly, to use SDA on an I64 system, you must familiarize yourself with the OpenVMS calling standard for I64 systems before you can decipher the pattern of a crash on the stack. For more information, refer to the *HP OpenVMS Calling Standard*.

4.5.2.2 Crash Log Utility Extractor

The Crash Log Utility Extractor (CLUE) is a tool for recording a history of crash dumps and key parameters for each crash dump and for extracting and summarizing key information. Unlike crash dumps, which are overwritten with each system crash and are available only for the most recent crash, the summary crash history file with a separate listing file for each crash is a permanent record of system crashes.

For this release, some of the CLUE commands are not yet available. For more information, refer to the *HP OpenVMS Industry Standard 64 for Integrity Servers Version 8.1 New Features and Release Notes*.

4.6 Testing the Migrated Application

You must test your application to compare the functionality of the migrated version with that of the Alpha version.

The first step in testing is to establish baseline values for your application by running your test suite on the Alpha application, as described in Section 4.3.

Once your application is running on an I64 system, there are two types of tests you will want to apply:

- The standard tests used for the Alpha version of the application
- New tests to check specifically for problems due to the change in architecture

4.6.1 Alpha Tests Ported to I64

Because the changes in your application are combined with use of a new architecture, testing your application after it is migrated to OpenVMS I64 is particularly important. Not only can the changes introduce errors into the application, but the new environment may bring out latent problems in the Alpha version.

Testing your migrated application involves the following steps:

1. Get a complete set of standard data for the application prior to the migration.
2. Migrate your test suite along with the application (if the tests are not already available on I64).
3. Validate the test suite on an I64 system.

Migrating Source Modules

4.6 Testing the Migrated Application

4. Run the migrated tests on the migrated application.

Both regression tests and stress tests are useful here. Stress tests are important to test for platform differences in synchronization, particularly for applications that use multiple threads of execution.

4.6.2 New I64 Tests

While your standard tests should go a long way toward verifying the function of the migrated application, you should add some tests that look at issues specific to the migration. Points to focus on include the following:

- Compiler differences
- Architectural differences
- Integration, such as modules written in different languages

4.6.3 Uncovering Latent Bugs

Despite your best efforts, and following all the previous suggestions, you may encounter bugs that were in your program all along, but never caused a problem on an OpenVMS Alpha system.

For example, a failure to initialize some variable in your program might have been benign on an Alpha computer but could produce an arithmetic exception on an I64 computer. The same could be true moving between any other two architectures, because the available instructions and the way compilers optimize them is bound to change. There is no magic answer for bugs that have been in hiding, but you should test your programs after porting them before making them available to other users.

4.7 Integrating the Migrated Application into a Software System

After you have migrated your application, check for problems that are caused by interactions with other software and that may have been introduced during the migration.

Sources of problems in interoperability can include the following:

- Alpha and I64 systems within an OpenVMS Cluster environment must use separate system disks. You must make sure that your application refers to the appropriate system disk.

- Image names

In a mixed environment, be sure that your application refers to the correct version.

- Native Alpha and native I64 versions of an image have the same name.

If you attempt to run an Alpha image on an I64 system, the following error message is displayed:

```
$ run alpha_image.exe
%DCL-W-ACTIMAGE, error activating image alpha_image.exe
-CLI-E-IMGNAME, image file alpha_image.exe
-IMGACT-F-NOT_I64, image is not an HP OpenVMS Industry Standard 64 image
$
```

4.8 Modifying Certain Types of Code

Certain coding practices and types of code will require changes. The coding practices and types of code that may or will require changes are:

- Alpha Macro 64 Assembler code.
This code must be rewritten in another language.
- Code that has been conditionalized for running on Alpha or VAX systems.
This code must be revised to express an I64 condition.
- Code that uses OpenVMS system services that have dependencies on the Alpha architecture.
- Code with other dependencies on the Alpha architecture.
- Code that uses floating point data types.
- Code that uses threads, in particular, custom-written tasking or stack switching.
- Privileged code.

4.8.1 Conditionalized Code

This section describes how to conditionalize OpenVMS code when migrating to I64. This code will be compiled for both Alpha and I64, or for VAX, Alpha, and I64. The symbol ALPHA, referred to in the following sections, is new. However, the symbol EVAX has not been eliminated. Do not feel obligated to replace EVAX with ALPHA but feel free to do so if convenient. The architecture symbols available for MACRO and BLISS are VAX, EVAX, ALPHA, and I64.

4.8.1.1 MACRO Sources

For MACRO-32 source files, the architecture symbols are in ARCH_DEFS.MAR, which is a prefix file specified on the command line. On Alpha, ALPHA and EVAX equal 1 while VAX and I64 are undefined. On I64, I64 equals 1 while VAX, EVAX, and ALPHA are undefined.

The following example show how to conditionalize MACRO-32 source code so that it can run on both Alpha and I64 systems.

For Alpha-specific code:

```
.IF DF ALPHA
    .
    .
    .
.ENDC
```

For I64-specific code:

```
.IF DF I64
    .
    .
    .
.ENDC
```

Migrating Source Modules

4.8 Modifying Certain Types of Code

4.8.1.2 BLISS Sources

For BLISS source files, either BLISS-32 or BLISS-64, the macros VAX, EVAX, ALPHA and I64 are defined in ARCH_DEFS.REQ. On Alpha, EVAX and ALPHA equal 1 while VAX and I64 equal 0. On I64, I64 equals 1 while VAX, EVAX, and ALPHA equal 0. You must require ARCH_DEFS.REQ to use these symbols in BLISS conditionals.

Note

The constructs %BLISS(*xxx*), %TARGET(*xxx*), and %HOST(*xxx*) are not recommended. Do not feel obligated to replace them, but feel free to do so if convenient.

Include the following statement in your source file:

```
REQUIRE 'SYS$LIBRARY:ARCH_DEFS';
```

Use the following statements in your source file to conditionalize code so that it can run on both Alpha and I64 systems.

For Alpha-specific code:

```
%if ALPHA %then
    .
    .
    .
%fi
```

For I64 specific code:

```
%if I64 %then
    .
    .
    .
%fi
```

4.8.1.3 C Sources

For C source files, the symbols __alpha, __ALPHA, __ia64, and __ia64__ are provided by the compilers on the appropriate platforms. Note that symbols could be defined on the compile command line but that is not the recommended method, nor is using arch_defs.h. Using #ifdef is considered the standard C programming practice.

For Alpha specific code, use the following:

```
#ifdef __alpha
    .
    .
    .
#endif
```

For I64 specific code, use the following:

```
#ifdef __ia64
    .
    .
    .
#endif
```

4.8.1.4 Existing Conditionalized Code

Existing conditional code must be examined to determine if changes are required. Here is an example of BLISS code to consider.

```
%IF VAX %THEN
    VVV
    VVV
%FI

%IF EVAX %THEN
    aaa
    aaa
%FI
```

If the code is truly architecture specific and you are adding I64 code, then the third case to add is:

```
%IF I64 %THEN
    iii
    iii
%FI
```

However, if the existing VAX/EVAX conditionals really reflect 32 bits versus 64 bits or an “old” versus “new” OpenVMS convention (for example, a promoted data structure or different routine to call) then the following method for conditionalizing code might be more appropriate. That is because Alpha and I64 code will be the same and that 64-bit code will need to be distinguished from the VAX code.

```
%IF VAX %THEN
    VVV
    VVV
%ELSE
    aaa
    aaa
%FI
```

4.8.2 System Services With Alpha Architecture Dependencies

Certain system services that work well in applications on OpenVMS Alpha do not port successfully to I64. These system services and their replacement services for I64 follow.

4.8.2.1 SYS\$GOTO_UNWIND

For OpenVMS Alpha, the SYS\$GOTO_UNWIND system service accepts a 32-bit invocation context handle by reference. You must change instances of this system service to SYS\$GOTO_UNWIND_64, which accepts a 64-bit invocation context. Make sure to alter source code to allocate space for the 64-bit value. Also, there are different library routines that return invocation context handles for OpenVMS I64. For more information, see the *HP OpenVMS Calling Standard*.

SYS\$GOTO_UNWIND is most frequently used to support programming language features, so changes are mostly in compilers or runtime libraries, but any direct use of SYS\$GOTO_UNWIND will require change.

4.8.2.2 SYS\$LKWSET and SYS\$LKWSET_64

The SYS\$LKWSET and SYS\$LKWSET_64 system services have been modified. For more information, see Section 4.8.8.

Migrating Source Modules

4.8 Modifying Certain Types of Code

4.8.3 Code With Other Dependencies on the Alpha Architecture

This section describes coding practices on Alpha that produce different results on I64 and may require changes to your application.

4.8.3.1 Initialized Overlaid Program Sections

Initialized overlaid program sections are handled differently on I64 systems. On OpenVMS Alpha systems, different portions of an overlaid program section may be initialized by multiple modules. This is not allowed on OpenVMS I64 systems. Please refer to Section 5.3.1.2 for more information on this change in behavior.

4.8.3.2 Condition Handlers Use of SS\$_HPARITH

On OpenVMS Alpha, SS\$_HPARITH is signaled for a number of arithmetic error conditions. On OpenVMS I64, SS\$_HPARITH is never signaled for arithmetic error conditions; instead, the more specialized SS\$_FLTINV and SS\$_FLTDIV error codes are signaled on OpenVMS I64.

Update condition handlers to detect these more specialized error codes. In order to keep code common for both architectures, wherever the code refers to SS\$_HPARITH, extend it for OpenVMS I64 to also consider SS\$_FLTINV and SS\$_FLTDIV.

4.8.3.3 Mechanism Array Data Structure

The mechanism array data structure on OpenVMS I64 is very different from the one on OpenVMS Alpha. Code that examines the mechanism array for the Alpha return status register of R0 must be changed. The return status code RETVAL has been extended to represent the return status register on both Alpha and I64 platforms. For more information, refer to the Alpha Mechanism Vector Format section in the *HP OpenVMS Calling Standard*.

4.8.3.4 Reliance on Alpha Object File Format

If your code relies on the layout of Alpha object files, you will need to modify it, because the object file format produced on OpenVMS I64 systems is different.

The object file format conforms to the 64-bit version of the Executable and Linkable Format (ELF), as described in the *System V Application Binary Interface* draft of 24 April 2001. This document, published by Caldera, is available on their web site at:

<http://www.caldera.com/developers/gabi>

The object file format also conforms to the I64 specific extensions described in the *Intel® Itanium™ Processor-specific Application Binary Interface (ABI)* May 2001 edition (document number 245270-003). Extensions and restrictions, necessary to support object file features that are specific to the OpenVMS operating system, have been added.

The image file format conforms to the DWARF Version 3 industry standard, which is available at the following location:

<http://www.eagercon.com/dwarf/dwarf3std.htm>

The debug symbol table representation on OpenVMS I64 is the industry-standard DWARF debug symbol table format described at this location. HP extensions to the DWARF Version 3 format will be published in a future release.

4.8.4 Code That Uses Floating Point Data Types

OpenVMS Alpha supports VAX floating point data types and IEEE floating point data types in hardware. OpenVMS I64 supports IEEE floating point in hardware and VAX floating-point data types in software.

The OpenVMS I64 compilers provide the `/FLOAT=D_FLOAT` and `/FLOAT=G_FLOAT` qualifiers to enable you to produce VAX floating point data types. If you do not specify one of these qualifiers, IEEE floating point data types will be used.

You can test an application's behavior with IEEE floating-point values on Alpha by compiling it with an IEEE qualifier on OpenVMS Alpha. If that produces acceptable results, you can build the application on an I64 system using the same qualifier.

When you compile an OpenVMS application that specifies an option to use VAX floating-point on I64, the compiler automatically generates code for converting floating-point formats. Whenever the application performs a sequence of arithmetic operations, this code does the following:

1. Converts VAX floating-point formats to either IEEE single or IEEE double floating-point formats.
2. Performs arithmetic operations in IEEE floating-point arithmetic.
3. Converts the resulting data from IEEE formats back to VAX formats.

Where no arithmetic operations are performed (VAX float fetches followed by stores), conversions do not occur. The code handles such situations as moves.

In a few cases, arithmetic calculations might have different results because of the following differences between VAX and IEEE formats:

- Values of numbers represented
- Rounding rules
- Exception behavior

These differences might cause problems for applications with certain characteristics.

For more information about the differences between floating point data types on OpenVMS Alpha and OpenVMS I64 and how these differences might affect ported applications, refer to Chapter 5 for the compiler or compilers you are using and to the "OpenVMS floating-point arithmetic on the Intel Itanium Architecture" white paper. You can obtain it from the following location:

http://www.hp.com/products1/evolution/alpha_retaintrust/openvms/resources.html

Note

Since the floating point white paper was written, the default `/IEEE_MODE` has changed from `FAST` to `DENORM_RESULTS`. This means that, by default, floating point operations may silently generate values that print as Infinity or Nan (the industry-standard behavior) instead of issuing a fatal run-time error as they would using VAX format float or `/IEEE_MODE=FAST`. Also, the smallest-magnitude non-zero value in this mode is much smaller because results are permitted to enter the denormal range instead of being flushed to zero as soon as the value is too

Migrating Source Modules

4.8 Modifying Certain Types of Code

small to represent with normalization. This default is the same default established by I64 at process startup.

4.8.4.1 LIB\$WAIT Problem and Solution

The use of LIB\$WAIT in code ported to OpenVMS I64 can cause unexpected results. An example in C follows.

```
float wait_time = 2.0;
lib$wait(&wait_time);
```

On OpenVMS I64 systems, this code sends an S_FLOATING into LIB\$WAIT. LIB\$WAIT expects an F_FLOATING, and gives a FLTINV exception.

LIB\$WAIT can accept three arguments. The previous code sequence can be rewritten with LIB\$WAIT using three arguments to run correctly on both I64 and Alpha systems. The following revised code works correctly when compiled without the /FLOAT qualifier:

```
#ifdef __ia64
    int float_type = 4; /* use S_FLOAT for I64 */
#else
    int float_type = 0; /* use F_FLOAT for Alpha */
#endif
float wait_time = 2.0;
lib$wait(&wait_time,0,&float_type);
```

A better coding method is to include LIBWAITDEF (from SYS\$STARLET_C.TLB) in your application and then specify the floating point data types by name. This code can be maintained more easily.

LIBWAITDEF includes the following symbols:

- LIB\$K_VAX_F
- LIB\$K_VAX_D
- LIB\$K_VAX_G
- LIB\$K_VAX_H
- LIB\$K_IEEE_S
- LIB\$K_IEEE_T

The following example shows how to include libwaitdef.h in the code and how to specify the floating-point data type names. This example also assumes that the program is not compiled with /FLOAT.

```
#include <libwaitdef.h>
.
.
.
#ifdef __ia64
    int float_type = LIB$K_IEEE_S; /* use S_FLOAT for IPF */
#else
    int float_type = LIB$K_VAX_F; /* use F_FLOAT for Alpha */
#endif
float wait_time = 2.0;
lib$wait(&wait_time,0,&float_type);
```

4.8.5 Code That Uses Threads

OpenVMS I64 supports all the thread interfaces that have been supported on OpenVMS since thread support was first introduced. Most OpenVMS Alpha code that uses threads can be ported to OpenVMS I64 without change. The exceptions are described in this section.

The major porting issue for code that uses threads is the usage of stack space. I64 code uses much more stack space than does equivalent Alpha code. Therefore, a threaded program that works fine on Alpha might get stack overflow failures on I64. For OpenVMS I64 Version 8.1, stack overflows typically appear as ACCVIOs, where the faulting virtual address is within a few pages of the value of R13 (the thread pointer register in the OpenVMS I64 calling standard). The goal for future releases of OpenVMS I64 is to have most stack overflows reported as STKOVF conditions (as they typically are on Alpha).

The default stack size is larger on OpenVMS I64 to help alleviate overflow problems. If the application requests a specific stack size, then the change to the default will be irrelevant, and the application source code may need to be changed to request more stack. HP recommends starting with an increase of three 8 Kb pages (24576 bytes). More or less may be needed.

Another side effect of the increased stack size requirement is increased demand on the P0 address region. Thread stacks are allocated from the P0 heap. Larger stacks may cause the process to exceed its memory quotas. In an extreme case, the P0 region could fill completely, in which case the process may need to reduce the number of threads in use concurrently (or make other changes to lessen the demand for P0 memory).

HP recommends that you familiarize yourself with the most recent improvements to thread support in OpenVMS, as documented in the *HP OpenVMS Alpha Version 7.3-2 Release Notes*. One change to the POSIX Threads C language header file `PTHREAD_EXCEPTION.H` caused problems when porting an application that relied on its former behavior.

For OpenVMS I64 Version 8.1, the following thread features are not supported:

- **THREADCP command.**

There is no equivalent to the OpenVMS Alpha command `THREADCP` (documented in the *Guide to the POSIX Threads Library*) in this release; a similar command is planned for a future OpenVMS I64 release. If you want to change the setting of the use-upcalls image flag, you need to relink the image.

- **Multiple kernel threads**

Support for multiple kernel threads is planned for a future release. However, the default for `MULTITHREAD` is 1, which allows use of upcalls but disallows use of multiple kernel threads. The lack of support for multiple kernel threads in this release should not have a significant effect for porting, but it will result in a performance limit.

Migrating Source Modules

4.8 Modifying Certain Types of Code

4.8.5.1 Thread Routines `cma_delay` and `cma_time_get_expiration`

Two legacy threads API library routines, `cma_delay` and `cma_time_get_expiration`, accept a floating point format parameter using the VAX F FLOAT format. Any application modules that call either of these routines must be compiled with either the `/FLOAT=D_FLOAT` or the `/FLOAT=G_FLOAT` qualifier to get VAX F FLOAT support. (However, if your application also uses double precision binary data, then you must use the `/FLOAT=G_FLOAT` qualifier.) For more information about floating point support, consult your compiler's documentation.

Note

These routines are no longer documented in user documentation but are still supported for use in legacy applications.

4.8.6 Code With Unaligned Data

HP recommends that you align your data naturally to achieve optimal performance for data referencing. Unaligned data can seriously degrade performance on both OpenVMS Alpha and OpenVMS I64.

Data is **naturally aligned** when its address is an integral multiple of the size of the data in bytes. For example, a longword is naturally aligned at any address that is a multiple of 4, and a quadword is naturally aligned at any address that is a multiple of 8. A structure is naturally aligned when all its members are naturally aligned.

Because natural alignment is not always possible, OpenVMS I64 systems provide help to manage the impact of unaligned data references. Alpha and I64 compilers automatically correct most potential alignment problems and flag others.

In addition to performance degradation, unaligned shared data can cause a program to execute incorrectly. Therefore, you must align shared data naturally. Shared data might be between threads of a single process, between a process and ASTs, or between several processes in a global section.

Finding the Problem

To find instances of unaligned data, you can use a qualifier provided by most I64 compilers that allows the compiler to report compile-time references to unaligned data.

Additional assistance, such as an OpenVMS Debugger qualifier to reveal unaligned data at run time, is planned for a future release.

Eliminating the Problem

To eliminate unaligned data, you can use one or more of the following methods:

- Compile with natural alignment, or, when language semantics do not provide for this, move data to be naturally aligned. Where filler is inserted to ensure that data remains aligned, there is a penalty in increased memory size. A useful technique for ensuring naturally aligned data while conserving memory is to declare longer variables first.
- Use high-level-language instructions that force natural alignment within data structures.

- Align data items on quadword boundaries.

Note

Software that is converted to natural alignment may be incompatible with other software that is running translated in the same OpenVMS Cluster environment or over a network. For example:

- An existing file format may specify records with unaligned data.
- A translated image may pass unaligned data to, or expect it from, a native image.

In such cases, you will have to adapt all parts of the application to expect the same type of data, either aligned or unaligned.

4.8.7 Code That Relies on the OpenVMS Alpha Calling Standard

If your application relies explicitly on characteristics of the OpenVMS Alpha calling standard, you will likely have to change it. The OpenVMS I64 calling standard is based on the Intel calling standard with some OpenVMS modifications. Significant differences introduced in the OpenVMS I64 calling standard include the following:

- No frame pointer (FP)
- Multiple stacks
- Only four registers preserved across calls
- Register numbers you are familiar with have changed

For more information, see Chapter 2.

4.8.8 Privileged Code

This section describes categories of privileged code that will require examination, and, in most cases, modifications.

4.8.8.1 Use of SYSS\$LKWSET and SYSS\$LKWSET_64

If your application uses SYSS\$LKWSET or SYSS\$LKWSET_64 to lock itself into memory, and your application does not run on VAX, consider replacing these calls with calls to the new LIB\$LOCK_IMAGE RTL routine. Similarly, replace the SYSS\$ULWSET and SYSS\$ULWSET_64 calls with calls to the new LIB\$UNLOCK_IMAGE RTL routine.

Programs that enter kernel mode and raise IPL above 2 must lock program code and data in the working set. Locking code and data is necessary to avoid crashing the system with a PGFIPLHI bugcheck.

On VAX, typically only the code and data explicitly referenced by the program need to be locked. On Alpha, the code, data and linkage data referenced by the program need to be locked. On I64, code, data, short data, and linker generated code need to be locked. To make porting easier and because the addresses of short data and linker generated data cannot be easily found within an image, changes have been made to the SYSS\$LKWSET and SYSS\$LKWSET_64 system services on Alpha and I64.

Migrating Source Modules

4.8 Modifying Certain Types of Code

The SYS\$LKWSET and SYS\$LKWSET_64 system services as of Version 8.1 test the first address passed in. If this address is within an image, these services attempt to lock the entire image in the working set. If a successful status code is returned, the program can raise IPL above 2 and be assured that it will not crash the system with a PGFIPLHI bugcheck.

A counter is maintained within the internal OpenVMS image structures that counts the number of times the image has been successfully locked in the working set. The counter is incremented when locked and decremented when unlocked. When the counter becomes zero, the entire image is unlocked from the working set.

If your privileged program runs on Alpha and I64 and not VAX, you can remove all the code that finds the code, data and linkage data and locks these areas in the working set. You can replace this code with calls to LIB\$LOCK_IMAGE and LIB\$UNLOCK_IMAGE. These routines are simpler to program correctly and make your code easier to understand and maintain.

If the program's image is too large to be locked in the working set, the status SS\$_LKWSETFUL is returned. If you encounter this status, you can increase the user's working set quota. Otherwise, you can split the image into two parts, one that contains the user mode code and another sharable image that contains the kernel mode code. At the entry to a kernel mode routine, the routine should call LIB\$LOCK_IMAGE to lock the entire image in the working set. Before exiting the kernel mode routine, the routine should call LIB\$UNLOCK_IMAGE.

4.8.8.2 Terminal Drivers

The interface for terminal class drivers on OpenVMS I64 is a call-based interface. This is a significant difference from the JSB-based interface on OpenVMS Alpha that uses registers to pass arguments.

The interface for OpenVMS I64 terminal class drivers is documented in the *OpenVMS Terminal Driver Port Class Interface for Itanium*. This document is available at the following location:

http://www.hp.com/products1/evolution/alpha_retaintrust/openvms/resources

OpenVMS I64 Development Environment

This chapter contains information about the OpenVMS I64 development environment, including:

- Native I64 Compilers
- Other development tools
- Linker
- Debugger
- Librarian utility

5.1 Native I64 Compilers

For OpenVMS I64, native I64 compilers are available for the following languages:

- Bliss
- C V6.5
- COBOL V2.8
- Fortran V7.5 (Fortran 90)
- Java (classic VM with interpreter, only)
- VAX MACRO-32

Additional native I64 compilers will be available in the future. The following native I64 compilers will be available in the future:

- BASIC
- C++
- Pascal

No support is provided to compile native Alpha Macro 64 assembler code to run on OpenVMS I64.

The I64 compilers provide command-line qualifiers to use VAX floating-point data types. For more information about these compilers, except for the VAX MACRO-32 compiler, see Chapter 6.

5.1.1 VAX MACRO-32 Compiler for OpenVMS I64

For new program development for OpenVMS I64, HP recommends the use of high-level languages. The VAX MACRO-32 Compiler for OpenVMS I64 is provided to convert existing VAX MACRO code into machine code that runs on OpenVMS I64 systems.

OpenVMS I64 Development Environment

5.1 Native I64 Compilers

Most VAX MACRO code can be compiled without any changes. The exceptions are:

- Programs that call routines that do not comply to the OpenVMS I64 calling standard.
- Programs that use the JSB instruction to call routines written in a language other than MACRO-32.

For more information on porting VAX MACRO programs to OpenVMS I64 systems, see the *HP OpenVMS MACRO Compiler Porting and User's Guide*.

5.2 Other Development Tools

Several other tools in addition to the compilers are available to develop, debug, and deploy native I64 applications:

- **OpenVMS Linker**
The OpenVMS Linker accepts I64 object files to produce an I64 image. For more information about the OpenVMS Linker, see Section 5.3.
- **OpenVMS Debugger**
The OpenVMS Debugger running on OpenVMS I64 has the same command interface as the current OpenVMS Alpha debugger. The graphical interface on OpenVMS Alpha systems will be available in a later release. For more information about the OpenVMS Debugger on OpenVMS I64, see Section 5.4.
- **XDelta Debugger**
The XDelta Debugger is an address location debugger that is used for debugging programs that run in privileged processor mode or at an elevated interrupt priority level. The XDelta Debugger is available for this release but the related Delta Debugger is not yet available.
- **OpenVMS Librarian utility**
The OpenVMS Librarian utility creates I64 libraries.
- **OpenVMS Message utility**
The OpenVMS Message utility allows you to supplement the OpenVMS system messages with your own messages.
- **ANALYZE/IMAGE**
The Analyze/Image utility can analyze I64 images.
- **ANALYZE/OBJECT**
The Analyze/Object utility can analyze I64 objects.
- **DECset**
DECset, a comprehensive set of development tools, includes the Language Sensitive Editor (LSE), the Digital Test Manager (DTM), Code Management System (CMS), and Module Management System (MMS). The two remaining DECset tools, Performance and Coverage Analyzer (PCA) and the Source Code Analyzer (SCA), will be available in a future release.
- **Command Definition utility**
- **System Dump Analyzer (SDA)**
SDA has been extended to display information specific to OpenVMS I64 systems.

- Crash Log Utility Extractor (CLUE)

5.2.1 Binary Translator

The binary translator is in development and will be available in a future release to translate Alpha user code images to run on OpenVMS I64 systems. The binary translator is useful for those cases where the source code is no longer available, or the compiler is not available on I64 (such as Fortran 77), or a third-party product has not been ported to OpenVMS I64. (Of course, the latter case would require obtaining permission from the owner of that software.) The binary translator will offer functionality similar to that of DECmigrate, a binary translator that was used to port OpenMVS VAX images to run on OpenVMS Alpha systems.

Note

The binary translator is not available with the Version 8.1 kit.

5.3 Linking Modules

The purpose of the linker is to create images, files that contain binary code and data. The linker ported to OpenVMS I64 is different from the linker on OpenVMS VAX and Alpha systems because it accepts OpenVMS I64 object files and produces OpenVMS I64 images. As on OpenVMS VAX and Alpha systems, the primary type of image created is an **executable image**. This image can be activated at the DCL command line by issuing the RUN command.

The OpenVMS I64 Linker also creates **shareable images**. A shareable image is a collection of procedures that can be called by executable images or other sharable images but are not executed using the RUN command. Rather, shareable images are included in an input file in a link operation in which an executable image is created.

When linking modules, the primary type of input file to the OpenVMS I64 Linker is the **object file**. Object files are produced by language processors such as compilers or assemblers. Because the object files produced by these compilers are unique to the Itanium® architecture, some aspects of linking modules on OpenVMS I64 are different. In general, the OpenVMS I64 linker interface as well as functional capabilities (for example, symbol resolution, virtual memory allocation, and image initialization) are similar to those on OpenVMS VAX and Alpha systems. This section provides an overview of the differences as well as considerations you should review before linking programs on OpenVMS I64 systems. These include:

- Differences when linking on I64 systems compared to VAX/Alpha systems (Section 5.3.1)
- Expanded linker map file information for I64 (Section 5.3.2)
- New linker qualifiers and options (link_qual_sec)

For details of these features and considerations, see the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes*.

5.3.1 Differences When Linking on OpenVMS I64 Systems

Although linking on OpenVMS I64 systems is similar to linking on OpenVMS Alpha systems, some differences exist. The following sections describe the key differences.

5.3.1.1 Mixed-Case Module Names Permitted in Link Operations

On OpenVMS I64 systems, object modules issued by some compilers (such as the Intel Assembler) can have module names containing mixed-case characters. Historically, these names have always been in uppercase. The linker will uppercase option file lines before processing them in order to obtain the uppercase named module from an object module library.

If a module with a mixed-case module name is to be explicitly included from an object module library, the linker option `CASE_SENSITIVE=YES` must be placed before a reference to the module name is made in the options file. See the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes* for more information and examples.

5.3.1.2 Handling of Initialized Overlaid Program Sections on OpenVMS I64

Handling of initialized overlaid program sections (PSECTS) produces different results on OpenVMS I64 systems than on Alpha systems. The condition occurs when two program sections, each declaring two longwords, are overlaid. The first program section initializes the first longword and the second program section initializes the second longword.

On OpenVMS Alpha systems, the linker produces an image section with the first and second longwords initialized. The linker uses the same start address for the program sections so they occupy the same virtual memory and allocate enough space to accommodate the largest of the program sections.

On OpenVMS I64 systems, the linker produces an image segment where either the first or second longword is initialized, depending on the order in which they were linked. Because the linker does not read all overlaid initialized program sections, and because these initializations can be made with identical values, the linker does not produce an error message for this condition. This will be corrected in a future release of the linker.

The linker will read the sections and will issue a warning when the initializations are different.

In the case of two or more initialized program sections with the same name and the overlay attribute, the linker issues the following error:

```
%ILINK-W-MULPSCINI, multiple initializations for overlaid psect psect name
in module name file name
```

The linker does not allow overlaid sections to be initialized with the same values. (On Alpha systems, overlaid sections could be initialized with the same value using an include file.) However, on I64 systems, the linker will copy a section and immediately perform relocation on it, making it difficult to compare the sections to determine if they are the same when they appear in multiple modules.

5.3.1.3 Conventions for Specifying Image Names

Images are given an image file specification (e.g., Foo.exe) that can be changed with the DCL RENAME command. In addition, an image name can be specified with the NAME=3D option and stored in the image in a note of type NT_VMS_IMGNAM. This name can be different than the image file specification name.

On Alpha systems, the image name specified with NAME =3D is used to identify self-references in the shareable image list (i.e., calls from within the image to itself, via an alias name in the symbol vector).

On I-64 systems, there is no entry in the shareable image list for the current image. Self-references are referred to with a special index value (-1 in the DT_VMS_FIXUP_NEEDED) into the shareable image list (i.e., a set of DT_NEEDED entries).

For complete information on these and other conventions for image names, see the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes*.

5.3.2 Expanded Map File Information

Information in the linker map file has been expanded for OpenVMS I64 systems:

- On Alpha systems, the map section that was titled Object Module Synopsis has been renamed on I64 systems to Object and Image Synopsis, and contains expanded object and image information.
- On Alpha systems, the map section titled Image Section Synopsis has been divided into two sections for I64 systems: Cluster Synopsis and Image Segment Synopsis. The information has been moved to the appropriate section.
- On I64 systems, the Image Segment Synopsis section replaces the Image Section Synopsis section in the Alpha linker map and shows each image segment as it was created.
- In the Program Synopsis Section, the PIC and NOPIC attributes, invalid on I64 systems, have been removed.
- In the Symbol Cross Reference section, the designation for an external symbol has changed. The prefix or suffix used on Alpha was RX, meaning relocatable and external. However, the linker does not know whether an external symbol is relocatable or not. As a result, on I64 systems the prefix or suffix has been changed to X (external).
- In the Symbols By Value section, Keys for Special Characters have been changed or expanded. For example, Unix weak symbols, designated by the UxWk key, are a new addition to OpenVMS I64.

For an example linker map illustrating this new information, see the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes*.

5.3.3 New Linker Qualifiers and Options for OpenVMS I64

Some new linker options and qualifiers have been added to support linking on OpenVMS I64 systems. This section describes these features.

OpenVMS I64 Development Environment

5.3 Linking Modules

5.3.3.1 New /BASE_ADDRESS Qualifier

The base address is the starting address that you want the linker to assign to an executable image. The linker assigns this address to the first program segment it creates. A new qualifier, /BASE_ADDRESS, has been provided on I64 systems. This qualifier is different from the BASE= option for VAX systems (which is illegal on Alpha and I64 systems), because it cannot be used to base shareable images. The /BASE_ADDRESS qualifier replaces the CLUSTER=[base-address] option on Alpha, which is also illegal on OpenVMS I64.

The /BASE_ADDRESS qualifier has the following format:

```
/BASE_ADDRESS=address
```

where address is the location at which you want the executable image based. You can express this location as decimal (%D), octal (%O), or hexadecimal (%X) numbers. The default is hexadecimal.

This qualifier works for all executable images and is accepted but ignored for shareable images, where the image activator determines the address of the shareable image at run time.

5.3.3.2 New Linker Option: SHORT_DATA

The OpenVMS I64 Linker provides a new SHORT_DATA option that allows you to combine read-only and read-write short data sections into a single segment.

Normally, there are two short data segments that the image activator relocates and combines. These segments are accessed by way of offsets to the GP register.

The SHORT_DATA=WRT option sets all short data program section attributes to WRT, including the linker-generated, read-only \$LINKER SDATA\$. This allows the linker to collect all short data program sections into a single, writable short data segment. In this case, \$LINKER SDATA\$ no longer starts at a page boundary and the linker reclaims unused space between the former read-only and read-write segments. When the short data sections are merged into a single segment, \$LINKER SDATA\$ is aligned on a 16 byte boundary.

Because short data segments are limited to 4 MBs, setting SHORT_DATA to WRT can eliminate the problem of overflow. Using this option reclaims the unused space between the two segments that can total up to 65,535 bytes.

The SHORT_DATA option has the following format:

```
SHORT_DATA=[NO]WRT
```

Where SHORT_DATA sets all short data program section attributes to WRT or NOWRT. This affects the collection of program sections, which are now placed in one writable or readonly short data segment.

caution

When setting SHORT_DATA to WRT, do not write to formerly read-only data because these data are no longer protected.

5.3.3.3 New alignments for the PSECT_ATTRIBUTE Option

The PSECT_ATTRIBUTE option now accepts integers 5, 6, 7, and 8 for the alignment attribute. The integers represent the byte alignment indicated as a power of 2. (For example, 2 ** 6 represents a 64-byte alignment.) The keyword HEXA (for hexadecimal word) was added for 2 ** 5 (i.e., a 32-byte or 16-word alignment).

5.4 Debugging Capabilities on OpenVMS I64 Systems

Several debuggers are provided on OpenVMS that allow for a wide range of debugging capabilities:

- OpenVMS Debugger (referred to simply as "debugger")
- Delta
- XDelta
- Source code debugger (SCD)

The OpenVMS Debugger and XDelta debugger are available on OpenVMS I64 Version 8.1. The following table shows the capabilities of these debuggers:

Category	OpenVMS Debugger	XDelta
Debugs operating systems	no	yes
Debugs applications	yes	no
Symbolic	yes	no
IPL greater than 0	no	yes
Process content	yes	no
User mode	yes	not†
Supervisor mode	no	not†
Exec mode	no	yes
Kernel mode	no	yes

†IPLs 0, 1 and 2 only

The following sections describe the capabilities of the OpenVMS Debugger and the XDelta debugger running on OpenVMS I64 systems. Further information about resolved problems, limitations, and known problems of the OpenVMS Debugger can be found in the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes*.

5.4.1 OpenVMS Debugger

The debugger provided on OpenVMS I64 is different from the OpenVMS VAX/Alpha debugger. Although the OpenVMS I64 debugger shares a similar command interface and debugging functionality, some differences do exist. The following sections describe the capabilities of the OpenVMS I64 Debugger and assume a familiarity with the debugger on OpenVMS VAX/Alpha systems.

OpenVMS I64 Development Environment

5.4 Debugging Capabilities on OpenVMS I64 Systems

5.4.1.1 Architecture Support

OpenVMS I64 Debugger supports the following hardware registers:

- General registers R0 through R127
- Floating registers F0 through F127
- Branch registers B0 through B7
- A 64-bit predicate value named PRED, representing predicate registers P0 through P63
- Application registers: AR16 (RSC), AR17 (BSP), AR18 (BSPSTORE), AR19 (RNAT), AR25 (CSD), AR26 (SSD), AR32 (CCV), AR36 (UNAT), AR64 (PFS), AR65 (LC), AR66 (EC)
- A program counter named PC, synthesized from the hardware IP register and the ri field of the PSR register
- Miscellaneous registers: CFM (current frame marker), UM (user mask), PSP (previous stack pointer), and IIPA (previously executed bundle address)

5.4.1.2 Language Support

OpenVMS I64 Debugger supports programs written in the following languages:

- Bliss
- C
- C++ (limited)
- COBOL (limited)
- IMACRO
- FORTRAN
- Intel® assembler (IAS)

Some issues exist when debugging supported languages on OpenVMS I64. See the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes* for information about these issues and suggested workarounds.

Support for C++ and COBOL in this release is limited. Generally, Debugger support for C++ is limited to programming constructs that are common to C, although support for basic C++ features such as references, classes and class members is available.

Do not use the SHOW SYMBOL/ADDRESS and SHOW SYMBOL/TYPE commands for C++ data declarations. Specific examples of these and other problems are described in the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes*.

IMACRO is a compiled language on OpenVMS I64. Because the Itanium® architecture has different hardware register usage than either Alpha or VAX, the IMACRO compiler must transform source code references to Alpha and VAX registers into a compatible register reference on Itanium. See the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes* for complete information on register mapping used by IMACRO.

OpenVMS I64 Development Environment

5.4 Debugging Capabilities on OpenVMS I64 Systems

5.4.1.3 Functional Areas and Commands

The following functional areas and commands are available:

- Screen mode: source display, instruction display, output display
- Instruction decoding
- Breakpoints
- Watchpoints (non-static)
- Step (all forms)
- \$ DEBUG/KEEP (kept debugger configuration)
- \$ RUN/DEBUG (normal debugger configuration)
- Multi-threaded programs
- CALL command
- Symbolic debugging of code in shareable images
- Symbolization of all possible static data locations
- Values in floating-point registers are displayed as IEEE T-floating values
- Examines and deposits of floating-point variables
- EXAMINE/ASCII command

5.4.1.4 Functionality Not Yet Ported

The following capabilities have not yet been ported to the OpenVMS I64:

- DECwindows Graphical User Interface
- Heap analyzer
- Screen mode register view
- SHOW STACK command

5.4.2 XDelta Debugger

In general, the XDelta Debugger for OpenVMS I64 systems behaves the same as XDelta on OpenVMS Alpha systems, with some restrictions. This section describes new capabilities added for OpenVMS and differences between XDelta on OpenVMS I64 and OpenVMS Alpha systems.

5.4.2.1 XDelta Capabilities on OpenVMS I64

New capabilities have been added for OpenVMS I64 systems:

- XDelta supports the following Itanium® registers:
 - General registers: R0 through R127
 - Floating registers: FP0 through FP127
 - Application registers: AR0 through AR127
 - Branch registers: BR0 through BR7
 - Control registers: CR0 through CR63
 - Miscellaneous registers: PC, PS, and CFM
 - Software implementation of Alpha registers
- ;D dump memory command

OpenVMS I64 Development Environment

5.4 Debugging Capabilities on OpenVMS I64 Systems

- ;T display interrupt stack command

For descriptions of the above registers and capabilities in this release, see the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes*. This document also describes the ;D and ;T commands, along with some restrictions.

5.4.2.2 Differences Between XDelta on OpenVMS I64 and OpenVMS Alpha Systems

This section describes the differences in XDelta behavior on I64 and Alpha systems.

Interrupting a Running System

To interrupt a running system on I64, press CTRL/P at the system console. Note that XDelta must have been previously loaded. When you press CTRL/P, the system halts at the current PC and at the current IPL. There is no delay in waiting for the IPL to drop below 14 as on Alpha systems.

Improved Symbolization

The X-registers are used by programmers to hold frequently-used values, such as base addresses of images and modules. When displaying breakpoints and other address values, XDelta now prints these values relative to the nearest X-register value. Previously, only certain values were checked for proximity to X-register values.

5.5 OpenVMS I64 Librarian Utility

The Librarian utility on OpenVMS I64 provides the same features provided by Librarian on OpenVMS Alpha, with some changes and restrictions. This section describes what is unique about the I64 Librarian. For information about restrictions or other temporary conditions, see the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes*.

5.5.1 Considerations When Using the I64 Librarian

You can use the DCL LIBRARY command (or Librarian LBR routines) to create libraries such as an I64 (ELF) Object library, I64 (ELF) Shareable image library, Macro library, Help library, and Text library. You can maintain the modules in a library or display information about a library and its modules. The I64 Librarian cannot create or process Alpha and VAX objects and shareable images. Rather, the default architecture for the Librarian is IA64.

There is no architecture switch for IA64 libraries. The Librarian works with OpenVMS ELF object and image libraries when used with the following qualifiers:

- /OBJECT - Uses OpenVMS ELF object libraries (default)
- /SHARE - Uses OpenVMS ELF shareable image libraries
- CREATE - Creates an OpenVMS ELF library of an object or shareable image type, depending whether /OBJECT or /SHARE qualifier is specified

The default library type created is an object library, if no OBJECT and SHARE qualifiers are specified.

5.5.2 Changes to the LBR\$ Routines

Two new library types for the LBR\$OPEN routine have been added on I64 systems:

- LBR\$C_TYP_ELFOBJ (9) - Represents an ELF object library
- LBR\$C_TYP_ELFHSTB (10) - Represents an ELF shareable image library

In addition, the following library types for the LBR\$OPEN routines are not supported in the I64 Librarian. You cannot use the library types to create or open OpenVMS Alpha or VAX object and shareable image libraries:

- LBR\$C_TYP_OBJ (1) - Represents a VAX object library
- LBR\$C_TYP_SHSTB (5) - Represents a VAX shareable image library
- LBR\$C_TYP_EOBJ (7) - Represents an Alpha object library
- LBR\$C_TYP_EHSTB (8) - Represents an Alpha shareable image library

5.5.3 I64 Library Format Handles Unix-Style Weak Symbols

Due to the requirements of the Intel C++ compiler, the I64 library format has been expanded to accommodate new Unix-style weak symbols. Multiple modules matching key names of new Unix-style weak symbols can now exist in the same library. The Librarian ignores the VMS-style weak symbol definitions as it has in the past.

Unix-style weak symbol definitions behave in the same manner as weak transfer addresses on OpenVMS; that is, their definitions are tentative. If a definition of a stronger binding type is not seen during a link operation, the tentative definition is designated as the definitive definition.

5.5.3.1 New ELF Type for Weak Symbols

A new Executable and Linkable Format (ELF) type was generated to distinguish between the two types of weak symbol definitions. For modules with ABI versions equal to 2 (the most common version used by compilers):

- Type STB_WEAK represents the Unix-style weak symbol (formerly, the VMS-style weak symbol definition for ABI Version 1 ELF format)
- Type STB_VMS_WEAK represents the VMS-style of weak symbol definition

The Librarian supports both the ELF ABI versions 1 and 2 of the object and image file formats within the same library.

Note

The new library format (version 4.0) only applies to ELF object and shareable image libraries. Other libraries remain at the version 3.0 format. Applications that reference the library via the currently-defined library services interface should not see any change in behavior.

5.5.3.2 Version 4.0 Library Index Format

HP recommends using the new Version 4.0 libraries. Older (Version 3.0) libraries can be opened by the Library Services but cannot be not modified. They can be converted to Version 4.0 libraries, with some restrictions. See the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes* for more information.

OpenVMS I64 Development Environment

5.5 OpenVMS I64 Librarian Utility

5.5.3.3 Current Library Limitation with Regard to Weak and Group Symbols

Library symbol entries are associated with the module within which they were defined. On OpenVMS I64, more than one module can define a Unix-style weak or Group symbol. As such, the I64 Librarian must keep an ordered list of defining modules. This list allows a previous association to be restored should the higher precedence association be removed (see more information on precedence ordering rules in the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes*).

Due to current limitations described in the *HP OpenVMS Industry Standard 64 Evaluation Release 8.1 New Features and Release Notes*, HP suggests that you only perform insert operations into the library for modules that contain Unix-style weak definitions. Should you decide to remove or replace modules in the library, you need to rebuild the library to make sure that the association of Unix-style weak definitions is correct.

5.5.3.4 New Group-Section Symbols

Symbols may be relative to sections contained in an ELF entity called a *group*. These groups, and the symbols associated with them, behave in a similar fashion as the new Unix-style weak symbol definitions; that is, they are tentative definitions. The librarian now allows multiple symbol definitions in the library's symbol name index.

Preparing to Port Applications

This chapter provides an overview of porting considerations related to the primary compilers that will be available for OpenVMS I64. Table 6–1 maps the versions of Alpha compilers that were used to port to the I64 versions.

Table 6–1 Mapping Alpha Compiler Versions to I64 Versions

Compiler	OpenVMS Alpha	OpenVMS I64	For more information
BASIC	V1.5	TBS	—
BLISS	V1.10-030	T1.1-049	See Section 6.1.
COBOL	V2.8-1286	T2.8-1340	See Section 6.2.
Fortran 77	—	Not available ¹	—
Fortran 90	V7.5	T8.0	See Section 6.3.
HP C	V6.5	T7.0	See Section 6.4.
HP C++	V6.5	TBS	—
Java	1.4.2-beta	1.4.2-beta1	See Section 6.5.
Macro-32	V4.1-18	T1.0-77	See Section 6.6.
Macro-64	V1.2	Not available ¹	—
Pascal	V5.8A	TBS	—

¹Compiler is platform-specific (for Alpha only).

Most of the OpenVMS I64 compilers share the following characteristics:

- They generate 32-bit code by default; you must use a compiler option to build a 64-bit program.
- The default is for IEEE floating datatypes instead of VAX floating datatypes. The white paper "OpenVMS Floating-Point Arithmetic on the Intel Itanium Architecture" describes how I64 deals with floating-point issues. You can link to this PDF-format white paper and other OpenVMS-to-Itanium architecture resources from the following web page:

http://www.hp.com/products1/evolution/alpha_retaintrust/openvms/resources.html

HP recommends that customers first compile their applications on OpenVMS Alpha with the versions of the compilers that are being ported to OpenVMS I64 in order to shake out any problems that might result from these newer compilers. (Some newer versions of compilers apply a stricter interpretation of existing compiler standards or enforce newer, stricter standards.) Once an application has compiled without error on the new compiler on an OpenVMS Alpha system, it can be ported to OpenVMS I64.

Preparing to Port Applications

6.1 Bliss Compiler

6.1 Bliss Compiler

This section describes the differences between the OpenVMS Alpha and OpenVMS I64 BLISS compilers.

BLISS-32EN and BLISS-64EN are native compilers running on and generating code for OpenVMS for Alpha systems.

BLISS-32IN and BLISS-64IN are native compilers running on and generating code for OpenVMS I64 systems.

The BLISS-32xx compilers perform operations 32 bits wide (that is, BLISS values are longwords). The default width is 32 bits. In this chapter, they are collectively referred to as "the 32-bit compilers."

The BLISS-64xx compilers perform operations 64 bits wide (that is, BLISS values are quadwords). The default width is 64 bits. In this chapter, they are collectively referred to as "the 64-bit compilers."

The compilers are invoked using the following commands:

Platform	Compiler	Command
Alpha	BLISS-32EN	BLISS/A32 or BLISS
Alpha	BLISS-64EN	BLISS/A64
I64	BLISS-32IN	BLISS/I32 or BLISS
I64	BLISS-64IN	BLISS/I64

6.1.1 Files

6.1.1.1 File Types

The default file type for object files for the OpenVMS compilers is .OBJ.

The default output file type for library files is .L32 for BLISS-32EN and BLISS-32IN, and .L64 for BLISS-64EN and BLISS-64IN. Library files are **not** compatible between dialects.

The search list for BLISS-32EN is as follows:

For source code: .B32E, .B32, .BLI
For require files: .R32E, .R32, .REQ
For library files: .L32E, .L32, .LIB

The search list for BLISS-64EN is as follows:

For source code: .B64E, .B64, .BLI
For require files: .R64E, .R64, .REQ
For library files: .L64E, .L64, .LIB

The search list for BLISS-32IN is as follows:

For source code: .B32I, .B32, .BLI
For require files: .R32I, .R32, .REQ
For library files: .L32I, .L32, .LIB

The search list for BLISS-64IN is as follows:

For source code: .B64I, .B64, .BLI
For require files: .R64I, .R64, .REQ
For library files: .L64I, .L64, .LIB

6.1.1.2 Output File Location Defaults

For the OpenVMS compilers, regardless of whether they run on Alpha or I64, the location of the output files depends on where in the command line the output qualifier was found.

In both Alpha and I64 BLISS, if an output file qualifier, such as /OBJECT, /LIST, or /LIBRARY, is used after an input file specification and does not include an output file specification, the output file specification defaults to the device, directory, and file name of the immediately preceding input file. For example:

```
$ BLISS /A32 [F00]BAR/OBJ      ! Puts BAR.OBJ in directory F00
$ BLISS /I32 [F00]BAR/OBJ      ! Puts BAR.OBJ in directory F00
$
$ BLISS /A32 /OBJ [F00]BAR      ! Puts BAR.OBJ in default directory
$ BLISS /I32 /OBJ [F00]BAR      ! Puts BAR.OBJ in default directory
$
$ BLISS /A32 [F00]BAR/OBJ=[]     ! Puts BAR.OBJ in default directory
$ BLISS /I32 [F00]BAR/OBJ=[]     ! Puts BAR.OBJ in default directory
```

6.1.2 Alpha BLISS Features Not Available

This section describes those Alpha BLISS features that are not supported by OpenVMS I64 BLISS.

6.1.2.1 Alpha BLISS Machine-Specific Builtins

Support for the following Alpha BLISS machine-specific builtins is being dropped:

- CMP_STORE_LONG (replaced by CMP_SWAP_LONG)
- CMP_STORE_QUAD (replaced by CMP_SWAP_QUAD)
- CMPBGE
- DRAINT
- RPCC
- TRAPB
- WRITE_MBX
- ZAP
- ZAPNOT

For information about CMP_SWAP_LONG and CMP_SWAP_QUAD, see Section 6.1.3.9.

6.1.2.2 Alpha BLISS PALcode Builtin Functions

Support for the following Alpha BLISS PALcode builtins is being dropped:

Preparing to Port Applications

6.1 Bliss Compiler

CALL_PAL	PAL_MFPR_PCBB	PAL_MTPR_SIRR
PAL_BPT	PAL_MFPR_PRBR	PAL_MTPR_SSP
PAL_BUGCHK	PAL_MFPR_PTBR	PAL_MTPR_TBIA
PAL_CFLUSH	PAL_MFPR_SCBB	PAL_MTPR_TBIAP
PAL_CHME	PAL_MFPR_SISR	PAL_MTPR_TBIS
PAL_CHMK	PAL_MFPR_SSP	PAL_MTPR_TBISD
PAL_CHMS	PAL_MFPR_TBCHK	PAL_MTPR_TBISI
PAL_CHMU	PAL_MFPR_USP	PAL_MTPR_USP
PAL_DRAIN	PAL_MFPR_VPTB	PAL_MTPR_VPTB
PAL_HALT	PAL_MFPR_WHAMI	PAL_PROBER
PAL_GENTRAP	PAL_MTPR_ASTEN	PAL_PROBEW
PAL_IMB	PAL_MTPR_ASTSR	PAL_RD_PS
PAL_LDQP	PAL_MTPR_DATFX	PAL_READ_UNQ
PAL_MFPR_ASN	PAL_MTPR_ESP	PAL_RSCC
PAL_MFPR_ASTEN	PAL_MTPR_FEN	PAL_STQP
PAL_MFPR_ASTSR	PAL_MTPR_IPIR	PAL_SWPCTX
PAL_MFPR_ESP	PAL_MTPR_IPL	PAL_SWASTEN
PAL_MFPR_FEN	PAL_MTPR_MCES	PAL_WRITE_UNQ
PAL_MFPR_IPL	PAL_MTPR_PRBR	PAL_WR_PS_SW
PAL_MFPR_MCES	PAL_MTPR_SCBB	PAL_MTPR_PERFMON

Macros will be placed in STARLET.REQ for PALCALL builtins. OpenVMS will supply the supporting code. The privileged CALL_PALs will call executive routines and the unprivileged CALL_PALs will execute system services.

6.1.2.3 Alpha BLISS Register Names

The following registers (IPF use is indicated) are not supported for naming in REGISTER, GLOBAL REGISTER, and EXTERNAL REGISTER, or as parameters to LINKAGE declarations.

R0	zero register
R1	global pointer
R2	volatile and GEM scratch register
R12	stack pointer
R13	thread pointer
R14-R16	volatile and GEM scratch registers
R17-R18	volatile scratch registers

6.1.2.4 Granularity of Byte and Word Stores and Fetches

Support for byte and word granularity of stores and fetches is being being dropped for OpenVMS I64 BLISS. This means that specifying any of the following will result in an error:

- The keyword BYTE to the /GRANULARITY qualifier.
- The switch DEFAULT_GRANULARITY=n where n is 0 or 1.
- The data attribute GRANULARITY=n where n is 0 or 1.

OpenVMS I64 BLISS continues to provide granularity support for LONGWORD and QUADWORD. See Section 6.1.3.7 for details.

6.1.2.5 INTERRUPT and EXCEPTION Linkages

INTERRUPT and EXCEPTION linkages are not supported.

6.1.2.6 BUILTIN Rn

You cannot specify an I64 register name to the BUILTIN keyword.

6.1.2.7 Link-Time Complex Expressions

Support for Link-Time Complex Expressions (LTCE) is not supported for OpenVMS I64 BLISS. The following example generates the same error for OpenVMS I64 BLISS as it would for TrueUnix64 Alpha BLISS:

```
global foobar: initial(foo - bar);
.....^
%BLS32-W-TEXT, Expression must be a link-time constant
at line number 5 in file d:[xxx]TESTLTCE.BLI;n
```

6.1.3 OpenVMS I64 BLISS Features

OpenVMS I64 BLISS provides only those existing Alpha BLISS features necessary to support OpenVMS I64. While Alpha BLISS enabled support for features of operating systems other than OpenVMS, such functionality is not included in the I64 BLISS compiler.

6.1.3.1 OpenVMS I64 BLISS Builtins

Only those builtins necessary for the correct operation of OpenVMS I64 are supported by the BLISS I64 compilers.

6.1.3.1.1 Common BLISS Builtins The following existing Common BLISS builtins are supported:

ABS	CH\$FIND_NOT_CH	CH\$WCHAR
ACTUALCOUNT	CH\$FIND_SUB	CH\$WCHAR_A
ACTUALPARAMETER	CH\$GEQ	MAX
ARGPTR	CH\$GTR	MAXA
BARRIER	CH\$LEQ	MAXU
CH\$ALLOCATION	CH\$LSS	MIN
CH\$A_RCHAR	CH\$MOVE	MINA
CH\$A_WCHAR	CH\$NEQ	MINU
CH\$COMPARE	CH\$PLUS	NULLPARAMETER
CH\$COPY	CH\$PTR	REF
CH\$DIFF	CH\$RCHAR	SETUNWIND
CH\$EQL	CH\$RCHAR_A	SIGN
CH\$FAIL	CH\$SIZE	SIGNAL
CH\$FILL	CH\$TRANSLATE	SIGNAL_STOP
CH\$FIND_CH	CH\$TRANSTABLE	

6.1.3.1.1.1 RETURNADDRESS Builtin

A new builtin function RETURNADDRESS will return the PC of the caller's caller.

This builtin takes no arguments and the format is:

```
RETURNADDRESS()
```

6.1.3.1.2 Machine-Specific Builtins The following Alpha BLISS machine-specific builtins are supported on the I64 BLISS compiler:

```
BARRIER
ESTABLISH
REVERT

ROT
SLL
SRA
SRL
UMULH

ADAWI
```

Preparing to Port Applications

6.1 Bliss Compiler

ADD_ATOMIC_LONG AND_ATOMIC_LONG OR_ATOMIC_LONG
ADD_ATOMIC_QUAD AND_ATOMIC_QUAD OR_ATOMIC_QUAD

XXX_ATOMIC_XXX no longer supports the optional retry-count input argument. See Section 6.1.3.5 for details.

TESTBITSSI TESTBITCC TESTBITCS
TESTBITCCI TESTBITSS TESTBITSC

TESTBITxx instructions no longer support the optional retry-count input argument or the optional success-flag output argument. See Section 6.1.3.6 for details.

ADDD	DIVD	MULD	SUBD	CMPD
ADDF	DIVF	MULF	SUBF	CMPF
ADDG	DIVG	MULG	SUBG	CMPG
ADD S	DIVS	MULS	SUBS	CMPS
ADD T	DIVT	MULT	SUBT	CMPT
CVTDF	CVTFD	CVTGD	CVTSF	CVTTD
CVTDG	CVTFG	CVTGF	CVTSI	CVTTG
CVTDI	CVTFI	CVTGI	CVTSL	CVTTI
CVTDL	CVTFL	CVTGL	CVTSQ	CVTTL
CVTDQ	CVTFQ	CVTGQ	CVTST	CVTTQ
CVTDT	CVTFS	CVTGT		CVTTS
CVTID	CVTLD	CVTQD		
CVTIF	CVTLF	CVTQF		
CVTIG	CVTLG	CVTQG		
CVTIS	CVTLS	CVTQS		
CVTIT	CVTLT	CVTQT		
CVTRDL	CVTRDQ			
CVTRFL	CVTRFQ			
CVTRGL	CVTRGQ			
CVTRSL	CVTRSQ			
CVTRTL	CVTRTQ			

6.1.3.1.3 New Machine-Specific Builtins A number of new builtins added to OpenVMS I64 BLISS provide access to single I64 instructions that can be used by the operating system.

6.1.3.1.3.1 Builtins for Single I64 Instructions

Each name that is capitalized in the following list is a new builtin function that can be specified. The lowercase name in parenthesis is the actual I64 instruction executed. The arguments to these instructions (and therefore their associated BLISS builtin names) are detailed in the *Intel IA-64 Architecture Software Developer's Manual*.

BREAK	(break)	LOADRS	(loadrs)	RUM	(rum)
BREAK2	(break)*	PROBER	(probe.r)	SRLZD	(srlz.d)
FC	(fc)	PROBEW	(probe.w)	SRLZI	(srlz.i)
FLUSHRS	(flushrs)	PCTE	(ptc.e)	SSM	(ssm)
FWB	(fwb)	PCTG	(ptc.g)	SUM	(sum)
INVALAT	(invala)	PCTGA	(ptc.ga)	SYNCI	(sync.i)
ITCD	(itc.d)	PTCL	(ptc.l)	TAK	(tak)
ITCI	(itc.i)	PTRD	(ptr.d)	THASH	(thash)
ITRD	(itr.d)	PTRI	(ptr.i)	TPA	(tpa)
ITRI	(itr.i)	RSM	(rsm)	TTAG	(ttag)

Note

The BREAK2 builtin requires two parameters. The first parameter, which must be a compile-time literal, specifies the 21-bit immediate value of the BREAK instruction. The second parameter, may be any expression whose

value is moved into IPF general register R17 just prior to executing the BREAK instruction.

6.1.3.1.3.2 Access to Processor Registers

The OpenVMS I64 BLISS compiler provides builtin functions for access to read and write the many and varied processor registers in the IPF implementations. The builtin functions are the following:

- GETREG
- SETREG
- GETREGIND
- SETREGIND

These builtins execute the mov.i instruction, which is detailed in the *Intel IA-64 Architecture Software Developer's Manual*. The two GET builtins return the value of the register specified.

To specify the register, a specially encoded integer constant is used, which is defined in an Intel C header file. See Appendix A of the *Intel IA-64 Architecture Software Developer's Manual* for the contents of this file.

6.1.3.1.4 PALcode Builtins Support for the following Alpha BLISS PALcode builtin functions has been retained:

PAL_INSQHIL	PAL_REMQHIL
PAL_INSQHILR	PAL_REMQHILR
PAL_INSQHIQ	PAL_REMQHIQ
PAL_INSQHIQR	PAL_REMQHIQR
PAL_INSQTIL	PAL_REMQTIL
PAL_INSQTILR	PAL_REMQTILR
PAL_INSQTIQ	PAL_REMQTIQ
PAL_INSQTIQR	PAL_REMQTIQR
PAL_INSQUEL	PAL_REMQUEL
PAL_INSQUEL_D	PAL_REMQUEL_D
PAL_INSQUEQ	PAL_REMQUEQ
PAL_INSQUEQ_D	PAL_REMQUEQ_D

Each of the 24 queue-manipulation PALcalls is implemented by BLISS as a call to an OpenVMS-provided SYS\$PAL_xxxx run-time routine.

6.1.3.2 BLI\$CALLG

The VAX idiom "CALLG(.AP, ...)" was replaced by an assembly routine BLI\$CALLG(ARGPTR(), .RTN) for OpenVMS Alpha BLISS. This routine as defined for OpenVMS ALpha BLISS has been rewritten for the I64 architecture and is supported for OpenVMS I64 BLISS.

6.1.3.3 I64 Registers

The I64 General registers, which can be named in REGISTER, GLOBAL REGISTER, and EXTERNAL REGISTER, and as parameters to LINKAGE declarations, are R3 through R11 and R19 through R31. In addition, parameter registers R32 through R39 can be named for parameters in LINKAGE declarations only.

Currently, there are no plans to support the naming of the I64 General registers R40 through R127.

Preparing to Port Applications

6.1 Bliss Compiler

Naming of any of the I64 Floating Point, Predicate, Branch and Application registers via the REGISTER, GLOBAL REGISTER, EXTERNAL REGISTER and LINKAGE declarations is not provided.

A register conflict message is issued when a user request for a particular register cannot be satisfied.

6.1.3.4 ALPHA_REGISTER_MAPPING Switch

A new module level switch, ALPHA_REGISTER_MAPPING, is being provided for OpenVMS I64 BLISS.

This switch can be specified in either the MODULE declaration or a SWITCHES declaration. Use of this switch causes a remapping of Alpha register numbers to I64 register numbers as described in this section.

Any register number specified as part of a REGISTER, GLOBAL REGISTER, EXTERNAL REGISTER or as parameters to GLOBAL, PRESERVE, NOPRESERVE or NOT USED in linkage declarations in the range of 0-31 are remapped according to the IMACRO mapping table as follows:

0 = GEM_TS_REG_K_R8	16 = GEM_TS_REG_K_R14
1 = GEM_TS_REG_K_R9	17 = GEM_TS_REG_K_R15
2 = GEM_TS_REG_K_R28	18 = GEM_TS_REG_K_R16
3 = GEM_TS_REG_K_R3	19 = GEM_TS_REG_K_R17
4 = GEM_TS_REG_K_R4	20 = GEM_TS_REG_K_R18
5 = GEM_TS_REG_K_R5	21 = GEM_TS_REG_K_R19
6 = GEM_TS_REG_K_R6	22 = GEM_TS_REG_K_R22
7 = GEM_TS_REG_K_R7	23 = GEM_TS_REG_K_R23
8 = GEM_TS_REG_K_R26	24 = GEM_TS_REG_K_R24
9 = GEM_TS_REG_K_R27	25 = GEM_TS_REG_K_R25
10 = GEM_TS_REG_K_R10	26 = GEM_TS_REG_K_R0
11 = GEM_TS_REG_K_R11	27 = GEM_TS_REG_K_R0
12 = GEM_TS_REG_K_R30	28 = GEM_TS_REG_K_R0
13 = GEM_TS_REG_K_R31	29 = GEM_TS_REG_K_R29
14 = GEM_TS_REG_K_R20	30 = GEM_TS_REG_K_R12
15 = GEM_TS_REG_K_R21	31 = GEM_TS_REG_K_R0

The mappings for register numbers 16-20, 26-28, and 30-31 translate into registers that are considered invalid specifications for OpenVMS I64 BLISS (see Section 6.1.2.3 and Section 6.1.3.3). Declarations that include any of these registers when ALPHA_REGISTER_MAPPING is specified will generate an error such as the following:

```
          r30 = 30,
          .....^
%BLS64-W-TEXT, Alpha register 30 cannot be declared, invalid mapping to
IPF register 12 at line number 9 in file ddd:[xxx]TESTALPHAREGMAP.BLI
```

Notice that the source line names register number 30 but the error text indicates register 12 is the problem. It is the translated register for 30, register 12, that is illegal to specify.

There is a special set of mappings for Alpha registers R16 through R21 if those registers are specified as linkage I/O parameters.

Note

For linkage I/O parameters *only*, the mappings for R16 through R21 are as follows:

```
16 = GEM_TS_REG_K_R32
17 = GEM_TS_REG_K_R33
18 = GEM_TS_REG_K_R34
19 = GEM_TS_REG_K_R35
20 = GEM_TS_REG_K_R36
21 = GEM_TS_REG_K_R37
```

6.1.3.4.1 /ANNOTATIONS Qualifier The OpenVMS I64 BLISS compiler supports a new compilation qualifier, /ANNOTATIONS. This qualifier provides information in the source listing regarding optimizations that the compiler is making (or not making) during compilation.

The qualifier accepts the following keywords that reflect the different listing annotations:

- ALL
- NONE
- CODE - Used for annotations of machine code listing. Currently, only NOP instructions are annotated.
- DETAIL - Provides greater detail; used in conjunction with other keywords.

The remaining keywords reflect GEM optimizations:

```
INLINING
LINKAGES
LOOP_TRANSFORMS
LOOP_UNROLLING
PREFETCHING
SHRINKWRAPPING
SOFTWARE_PIPELINING
TAIL_CALLS
TAIL_RECURSION
```

All keywords, with the exception of ALL and NONE, are negatable. The qualifier itself is also negatable. By default it is not present in the command line.

If the /ANNOTATIONS qualifier is specified without any parameters, the default is ALL.

6.1.3.4.2 /ALPHA_REGISTER_MAPPING Qualifier The OpenVMS I64 BLISS compiler supports a new compilation qualifier to enable ALPHA_REGISTER_MAPPING without having to modify the source. This is a positional qualifier. Specifying this qualifier on the compilation line for a module is equivalent to setting the ALPHA_REGISTER_MAPPING switch in the module header.

6.1.3.4.3 /ALPHA_REGISTER_MAPPING Informationals For OpenVMS I64 BLISS three new informational messages have been added.

- If the /ALPHA_REGISTER_MAPPING qualifier was specified on the command line the following message is displayed:
%BLS64-I-TEXT, Alpha Register Mapping enabled by the command line
- If the switch ALPHA_REGISTER_MAPPING is specified in the module header or as an argument to the SWITCH declaration the following message is displayed:

Preparing to Port Applications

6.1 Bliss Compiler

```
MODULE SIMPLE (MAIN=TEST, ALPHA_REGISTER_MAPPING)=  
.....^  
%BLS64-I-TEXT, Alpha Register Mapping enabled
```

- If the switch NOALPHA_REGISTER_MAPPING is specified in the module header or as an argument to the SWITCH declaration the following message is displayed:

```
MODULE SIMPLE (MAIN=TEST, NOALPHA_REGISTER_MAPPING)=  
.....^  
%BLS64-I-TEXT, Alpha Register Mapping disabled
```

6.1.3.5 ADD, AND, Builtin Functions for Atomic Operations

The ADD_ATOMIC_XXXX, AND_ATOMIC_XXXX, and OR_ATOMIC_XXXX builtin functions for atomic updating of memory are supported by OpenVMS I64 BLISS. They are listed in Section 6.1.3.1.2.

Because the I64 instructions to support these builtins waits until the operation succeeds, the optional retry-count input parameter has been eliminated. These builtins now have the form:

*option*_ATOMIC_size(*ptr*, *expr* [:*old_value*]) !Optional output

Where:

option can be AND, ADD, or OR.

size can be LONG or QUAD.

ptr must be a naturally aligned address.

Value can be 0 (Operation failed) or 1 (Operation succeeded)

The operation is addition (or ANDing or ORing) of the expression EXPR to the data-segment pointed to by PTR in an atomic fashion.

The optional output parameter OLD_VALUE is set to the previous value of the data segment pointed to by PTR.

Any attempt to use the OpenVMS Alpha BLISS optional *retry_count* results in a syntax error.

6.1.3.6 TESTBITxxI and TESTBITxx Builtin Functions for Atomic Operations

The TESTBITxxI and TESTBITxx builtin functions for atomic operations are supported by OpenVMS I64 BLISS. They are listed in Section 6.1.3.1.2.

Because the I64 instruction to support these builtins waits until the operation succeeds, the optional input parameter *retry_count* and the optional output parameter *success_flag* have been eliminated. These builtins now have the form:

TESTBITxxx(*field*)

Any attempt to use the OpenVMS Alpha BLISS optional *retry_count* or *success_flag* arguments results in a syntax error.

6.1.3.7 Granularity of Longword and Quadword Writes

OpenVMS I64 BLISS supports the /GRANULARITY=keyword qualifier, the switch DEFAULT_GRANULARITY=n, and the data attribute GRANULARITY(n) as described in this section.

Users can control the granularity of stores and fetches by using the command line qualifier /GRANULARITY=keyword, the switch DEFAULT_GRANULARITY=n, and the data attribute GRANULARITY(n).

The keyword in the command line qualifier must be LONGWORD, or QUADWORD. The parameter *n* must be either 2 (longWORD) or 3 (quadWORD).

When these are used together, the data attribute has the highest priority. The switch, when used in a SWITCHES declaration, sets the granularity of data declared after it within the same scope. The switch may also be used in the module header. The command line qualifier has the lowest priority.

6.1.3.8 Shift Builtin Functions

Builtin functions for shifts in a known direction are supported for OpenVMS I64 BLISS. They are listed in Section 6.1.3.1.2. These functions are valid only for shift amounts in the range 0..%BPVAL-1.

6.1.3.9 Compare and Swap Builtin Functions

Both OpenVMS Alpha and OpenVMS I64 BLISS provide support for the following new compare and swap builtin functions:

- CMP_SWAP_LONG(addr, comparand, value)
- CMP_SWAP_QUAD(addr, comparand, value)

These functions do the following interlocked operations: compare the longword or quadword at *addr* with *comparand*, and if they are equal, store *value* at *addr*. They return an indicator of success (1) or failure (0).

6.1.3.10 I64-Specific Multimedia Instructions

There are no plans to support access to the I64-specific multimedia-type instructions.

6.1.3.11 Linkages

6.1.3.11.1 CALL Linkage The CALL linkage, as described in this section for OpenVMS Alpha BLISS, is also supported by OpenVMS I64 BLISS.

Routines compiled with a 32-bit compiler can call routines compiled with a 64-bit compiler and vice versa. Parameters are truncated when shortened, and sign-extended when lengthened.

By default, CALL linkages pass an argument count. This can be overridden using the NOCOUNT linkage option.

Although the arguments are passed in quadwords, the 32-bit compilers can "see" only the lower 32 bits.

6.1.3.11.2 JSB Linkage The OpenVMS I64 BLISS compilers have a JSB linkage type. Routines declared with the JSB linkage comply with the JSB rules developed for OpenVMS I64.

6.1.3.12 /[NO]TIE Qualifier

Support for this qualifier continues for OpenVMS I64. /NOTIE is the default.

TIE is used to enable the compiled code to be used in combination with translated images, either because the code might call into a translated image or might be called from a translated image.

In particular, TIE does the following:

- Causes the inclusion of procedure signature information in the compiled program. This may increase the size and possibly also the number of relocations processed during linking and image activation, but does not otherwise affect performance.

Preparing to Port Applications

6.1 Bliss Compiler

- Causes calls to procedure values (sometimes called indirect or computed calls) to be compiled using a service routine (OTS\$CALL_PROC); this routine determines whether the target procedure is native IPF code or in a translated image and proceeds accordingly.

6.1.3.13 /ENVIRONMENT=([NO]FP) and ENVIRONMENT([NO]FP)

The /ENVIRONMENT=([NO]FP) qualifier and the ENVIRONMENT([NO]FP) switch were provided for OpenVMS Alpha BLISS to cause the compiler to disable the use of floating point registers for certain integer division operations.

For OpenVMS I64 BLISS, the /ENVIRONMENT=NOFP command qualifier or ENVIRONMENT(NOFP) switch does not totally disable floating point due to the architectural features of I64. Instead, source code is still restricted to not have floating point operations, but the generated code for certain operations (in particular, integer multiplication and division and the constructs that imply them) are restricted to use a small subset of the floating point registers. Specifically, if this option is specified, the compiler is restricted to using f6-f11, and will set the ELF EF_IA_64_REDUCEFP option described in Section 4.1.1.6 of the *Intel Itanium Processor-Specific Application Binary Interface*.

The /ENVIRONMENT=FP command qualifier and ENVIRONMENT(FP) switch are unaffected.

6.1.3.14 Floating Point Support

6.1.3.14.1 Floating Point Builtin Functions BLISS does not have a high level of support for floating-point numbers. The extent of the support involves the ability to create floating-point literals, and there are machine-specific builtins for floating-point arithmetic and conversion operations. For a complete list see Section 6.1.3.1.2.

None of the floating point builtin functions detect overflow, so they do not return a value.

6.1.3.14.2 Floating Point Literals The floating point literals supported by OpenVMS I64 BLISS are the same set supported by OpenVMS Alpha BLISS: %E, %D, %G, %S and %T.

6.1.3.14.3 Floating Point Registers Direct use of the I64 floating-point registers is not supported.

6.1.3.14.4 Calling Non-BLISS Routines with Floating Point Parameters

It is possible to call standard non-BLISS routines that expect floating-point parameters passed by value, and that return a floating-point or complex value.

The standard functions %FFLOAT, %DFLOAT, %GFLOAT, %SFLOAT and %TFLOAT are supported by OpenVMS I64 BLISS.

6.1.3.15 New and Expanded Lexicals

BLISS has added the following new compiler-state lexicals to support the OpenVMS I64 compilers: BLISS32I and BLISS64I.

- %BLISS now recognizes BLISS32V, BLISS32E, BLISS64E, BLISS32I and BLISS64I.

%BLISS(BLISS32) is true for all 32-bit BLISS compilers.

%BLISS(BLISS32V) is true only for VAX BLISS (BLISS-32).

%BLISS(BLISS32E) is true for all 32-bit Alpha compilers.

%BLISS(BLISS64E) is true for all 64-bit Alpha compilers.

%BLISS(BLISS32I) is true for all 32-bit I64 compilers.

%BLISS(BLISS64I) is true for all 64-bit I64 compilers.

- The lexicals %BLISS32I and %BLISS64I have been added. Their behavior parallels that of the new parameters to %BLISS.
- Support for the I64 architecture as a keyword to the %HOST and %TARGET lexicals has been added for OpenVMS I64 BLISS.

6.1.3.16 OpenVMS I64 BLISS Support for IPF Short Data Sections

The IPF calling standard requires that all global data objects with a size of 8 bytes or smaller be allocated in short data sections.

Short data sections can be addressed with an efficient code sequence that involves adding a 22-bit literal to the contents of the GP base register. This code sequence limits the combined size of all the short data sections. A linker error occurs if the total amount of data allocated to short data sections exceeds a size of 2^{22} bytes. Compilers on IPF can use GP relative addressing when accessing short globals and short externals.

OpenVMS I64 BLISS exhibits new behavior for the PSECT attribute GP_RELATIVE and for the new PSECT attribute SHORT, which supports allocating short data sections.

Specifying the GP_RELATIVE keyword as a PSECT attribute causes PSECT to be labeled as containing short data so that the linker will allocate the PSECT close to the GP base address.

The syntax of the SHORT attribute is as follows:

"SHORT" "(" psect-name ")"

The following rules apply to the SHORT attribute:

- If the psect-name in a SHORT attribute is not yet declared then its appearance in a SHORT attribute constitutes a declaration. The attributes of the PSECT containing the SHORT attribute become the attributes of the PSECT named in the SHORT attribute, except that the PSECT name declared in the SHORT attribute does not have the SHORT attribute and the PSECT name declared in the SHORT attribute does have the GP_RELATIVE attribute.
- If the psect-name in a SHORT attribute has been previously declared then its attributes are not changed. A warning message is generated if the PSECT named in a SHORT attribute does not have the GP_RELATIVE attribute.
- If a data object with storage class OWN, GLOBAL or PLIT has a size of 8 or fewer bytes and the data object is specified to be allocated to a PSECT that includes the SHORT attribute, then that object is allocated to the PSECT named in the SHORT attribute. Note that this is a one-step process that is not recursive. If a short data object has its allocation PSECT renamed by the SHORT attribute, then the SHORT attribute of the renamed PSECT is not considered for any further renaming.
- Data objects with sizes larger than 8 bytes ignore the SHORT attribute.
- Data objects in the CODE, INITIAL and LINKAGE storage classes ignore the SHORT attribute, regardless of their size.

Preparing to Port Applications

6.1 Bliss Compiler

- For the purposes of PSECT renaming by means of the SHORT attribute, the size of a PLIT object does not include the size of the count word that precedes the PLIT data.

Example

PSECT

```
NODEFAULT = $GLOBAL_SHORT$
  (READ,WRITE,NOEXECUTE,NOSHARE,NOPICT,CONCATENATE,LOCAL,ALIGN(3),
   GP_RELATIVE),
```

! The above declaration of \$GLOBAL_SHORT\$ is not needed. If the above
! declaration were deleted then the SHORT(\$GLOBAL_SHORT\$) attribute in
! the following declaration would implicitly make an identical
! declaration of \$GLOBAL_SHORT\$.

```
GLOBAL = $GLOBAL$
  (READ,WRITE,NOEXECUTE,NOSHARE,NOPICT,CONCATENATE,LOCAL,ALIGN(3),
   SHORT($GLOBAL_SHORT$)),
```

```
NODEFAULT = MY_GLOBAL
  (READ,WRITE,NOEXECUTE,SHARE,NOPICT,CONCATENATE,LOCAL,ALIGN(3)),
```

```
PLIT = $PLIT$
  (READ,NOWRITE,NOEXECUTE,SHARE,NOPICT,CONCATENATE,GLOBAL,ALIGN(3),
   SHORT($PLIT_SHORT$));
```

GLOBAL

```
  X1,                ! allocated in $GLOBAL_SHORT$
  Y1 : VECTOR[2, LONG], ! allocated in $GLOBAL_SHORT$
  Z1 : VECTOR[3, LONG], ! allocated in $GLOBAL$
  A1 : PSECT(MY_GLOBAL), ! allocated in MY_GLOBAL
  B1 : VECTOR[3, LONG] PSECT(MY_GLOBAL), ! allocated in MY_GLOBAL
  C1 : VECTOR[3, LONG]
        PSECT($GLOBAL_SHORT$); ! allocated in $GLOBAL_SHORT$
```

PSECT GLOBAL = MY_GLOBAL;

! use MY_GLOBAL as default for both noshort/short

GLOBAL

```
  X2,                ! allocated in MY_GLOBAL
  Y2 : VECTOR[2, LONG], ! allocated in MY_GLOBAL
  Z2 : VECTOR[3, LONG], ! allocated in MY_GLOBAL
  A2 : PSECT($GLOBAL$), ! allocated in $GLOBAL_SHORT$
  B2 : VECTOR[3, LONG] PSECT($GLOBAL$); ! allocated in $GLOBAL$;
```

! Note that the allocations of A1, X2 and Y2 violate the calling
! standard rules. These variables cannot be shared with other
! languages, such as C or C++.

PSECT GLOBAL = \$GLOBAL\$;

! back to using \$GLOBAL\$/\$GLOBAL_SHORT\$ as default noshort/short

GLOBAL BIND

```
  P1 = UPLIT("abcdefghi"), ! allocated in $PLIT$
  P2 = PLIT("abcdefgh"), ! allocated in $PLIT_SHORT$
  P3 = PSECT(GLOBAL) PLIT("AB"), ! allocated in $GLOBAL_SHORT$
  p4 = PSECT($PLIT_SHORT$)
        PLIT("abcdefghijklmn"), ! allocated in $PLIT_SHORT$
  P5 = PSECT(MY_GLOBAL) PLIT("AB"); ! allocated in MY_GLOBAL
```

Notes

- The allocations of A1, X2, Y2, and P5 violate the calling standard rules. These variables cannot be shared with other languages, such as C or C++. They can be shared with modules written in BLISS and MACRO.

- The current Openvms I64 BLISS design does not support using GP_RELATIVE addressing mode on EXTERNAL variable references. However, the usual GENERAL addressing mode used by EXTERNAL variables correctly references a GP_RELATIVE section. There are no current plans to add an ADDRESSING_MODE(GP_RELATIVE) attribute to BLISS.
-

6.2 COBOL

COBOL Version 2.8 is supported on both Alpha and I64. The main difference between these compilers is that the default is for IEEE floating point instead of VAX floating point, which is true for all the HP compilers.

See Section 6.2.1 and the *COBOL Release Notes* for restrictions and known problems related to using COBOL on OpenVMS I64.

6.2.1 /ARCH and /OPTIMIZE=TUNE Qualifiers

For the sake of "compile-and-go" compatibility, Alpha values for the /ARCH and /OPTIMIZE=TUNE qualifiers are accepted on the compiler invocation command. An informational message is printed saying they are ignored.

I64 values for /ARCH and /OPTIMIZE=TUNE are defined in the OpenVMS I64 compiler for development purposes only. Their behavior is unpredictable and they should not be used.

6.3 Fortran

Fortran 90 Version T8.0 is supported on OpenVMS I64. Fortran 77 is not supported on OpenVMS I64. (See Section 6.3.2 for details.)

HP Fortran for OpenVMS I64 Systems features the same command-line options and language features as HP Fortran for OpenVMS Alpha Systems with a few exceptions, which are discussed in the following sections.

6.3.1 Floating Point Arithmetic

The Fortran release notes and the floating point white paper mentioned at the beginning of this chapter together describe how HP Fortran for I64 deals with floating-point issues. Highlights are summarized as follows:

- IEEE is the default floating-point datatype (that is, the default is /FLOAT=IEEE_FLOAT).

The native Alpha compiler defaults to /FLOAT=G_FLOAT. On OpenVMS I64 systems, there is no hardware support for floating-point representations other than IEEE. The VAX floating-point formats are supported in the compiler by generating run-time code to convert to IEEE format before performing arithmetic operations, and then converting the IEEE result back to the appropriate VAX format. This imposes additional run-time overhead and some loss of accuracy compared to performing the operations in hardware on Alpha (and VAX).

The software support for the VAX formats is an important functional compatibility requirement for certain applications that need to deal with on-disk binary floating-point data, but its use should not be encouraged. This change is similar to the change in default from /FLOAT=D_FLOAT on VAX to /FLOAT=G_FLOAT on Alpha.

Preparing to Port Applications

6.3 Fortran

If at all possible, users should use `/FLOAT=IEEE_FLOAT` (the default) for the highest performance and accuracy.

Note that the meaning of `/CONVERT=NATIVE` has changed; on Alpha it meant VAX-format floating-point, while on OpenVMS I64 it means IEEE-format floating-point. Files written from Fortran applications built with `/CONVERT=NATIVE` format on Alpha may be read by I64 applications built with `/CONVERT=VAXG`.

- The `/IEEE_MODE` qualifier defaults to `/IEEE_MODE=DENORM_RESULTS`. This differs from Alpha, where the default is `/IEEE_MODE=FAST`. Despite the name, `/IEEE_MODE=FAST` does not have a significant effect on run-time performance on I64 (or on Alpha processors from EV6 onward).
- Now and in the future, users will have to pick one `/FLOAT` value and one `/IEEE_MODE` value and stick with it for the whole of their application. This is because mixed-mode applications will not (in general) work on OpenVMS I64 systems as a result of architectural differences in the hardware. This is a change from OpenVMS on Alpha systems, where mixed-mode applications work. In particular, per-routine/per-file/per-library settings of a mode will not work.
- If your code changes the floating-point exception mode, it is your responsibility to change it back on routine-exit, including exits by means of exception handlers. Failure to do so may lead to unexpected results, as the compilers will assume that no called routine changes the current mode. This is also a requirement on user-written libraries.
- Exception handlers will be entered with the floating-point mode in effect at the time the exception was raised, not the mode with which the handler was compiled.

There are still some open floating point implementation issues, particularly regarding handling of floating-point exceptions. The behavior of the compilers and operating system may change between this beta release and the final product.

6.3.2 Only the F90 Compiler is Supported

The F77 compiler, previously invoked with the `/OLD_F77` qualifier, is not available. Development is currently underway to provide the following functionality contained in the Alpha F77 compiler that is not available in the I64 and Alpha F90 compilers:

- FDML support has been implemented in F90, but it has not yet been fully tested.
- CDD support is planned in F90, but is not yet available.
- Code previously compiled with the F77 compiler that does not compile with the F90 compiler should be reported through the problem reporting mechanism for this beta test. Although it may not be possible to eliminate all differences, the development team will attempt to minimize these differences by enhancing the F90 compiler.

Note

The lack of support for `/OLD_F77` should not be confused with the `/F77` qualifier, which indicates that the compiler uses FORTRAN-

77 interpretation rules for those statements that have a meaning incompatible with FORTRAN-66, and which will be supported.

6.3.3 /ARCH and /OPTIMIZE=TUNE Qualifiers

For the sake of "compile-and-go" compatibility, Alpha values for the /ARCH and /OPTIMIZE=TUNE qualifiers are accepted on the compiler invocation command. An informational message is printed saying they are ignored.

I64 values for /ARCH and /OPTIMIZE=TUNE are defined in the OpenVMS I64 compiler for development purposes only. Their behavior is unpredictable and they should not be used.

6.4 HP C/ANSI C

HP C Version T7.0 is supported on I64.

6.4.1 I64 Floating Point Default

The native Alpha compiler defaults to /FLOAT=G_FLOAT. For I64, the default is /FLOAT=IEEE_FLOAT/IEEE=DENORM.

On OpenVMS I64, there is no hardware support for floating point representations other than IEEE. The VAX floating point formats are supported in the compiler by generating run-time code to convert to IEEE format in order to perform arithmetic operations, and then convert the IEEE result back to the appropriate VAX format. This imposes additional run-time overhead and a possible loss of accuracy compared to performing the operations in hardware on the Alpha (and VAX). The software support for the VAX formats is an important functional compatibility requirement for certain applications that deal with on-disk binary floating-point data, but its use should not be encouraged by letting it remain the default. This change is similar to the change in default from /FLOAT=D_FLOAT on VAX to /FLOAT=G_FLOAT on Alpha.

Note also that the default /IEEE_MODE has changed from FAST (on OpenVMS Alpha) to DENORM_RESULTS on OpenVMS I64. This means that, by default, floating point operations may silently generate values that print as Infinity or Nan (the industry-standard behavior) instead of issuing a fatal run-time error as they would using VAX format float or /IEEE_MODE=FAST. Also, the smallest-magnitude non-zero value in this mode is much smaller because results are permitted to enter the denormal range instead of being flushed to zero as soon as the value is too small to represent with normalization.

6.4.2 Semantics of /IEEE_MODE Qualifier

On OpenVMS Alpha, the /IEEE_MODE qualifier generally has its greatest effect on the generated code of a compilation. When calls are made between functions compiled with different /IEEE_MODE qualifiers, each function produces the /IEEE_MODE behavior with which it was compiled.

On OpenVMS I64, the /IEEE_MODE qualifier primarily affects only the setting of a hardware register at program startup. In general, the /IEEE_MODE behavior for a given function is controlled by the /IEEE_MODE option that was specified on the compilation that produced the main program. The startup code for the main program sets the hardware register according to the command line qualifiers used to compile the main program.

Preparing to Port Applications

6.4 HP C/ANSI C

When the `/IEEE_MODE` qualifier is applied to a compilation that does not contain a main program, the qualifier does have some effect: it can affect the evaluation of floating-point constant expressions, and it is used to set the `EXCEPTION_MODE` used by the math library for calls from that compilation.

The qualifier has no effect on the exceptional behavior of floating-point calculations generated as inline code for that compilation. Therefore, if floating point exceptional behavior is important to an application, all of its compilations, including the one containing the main program, should be compiled with the same `/FLOAT` and `/IEEE_MODE` settings.

Note that even on Alpha, setting `/IEEE_MODE=UNDERFLOW_TO_ZERO` requires the setting of a run-time status register; therefore, this setting needs to be specified on the compilation containing the main program in order to be effective in other compilations.

6.4.3 Predefined Macros

The compiler predefines a number of macros with the same meanings as in the native Alpha compiler, except that it does not predefine any of the macros that specify the Alpha architecture. Instead, it predefines the macros `__ia64` and `__ia64__`, as is the practice in the Intel and gcc compilers for I64. The change in floating-point representation from `G_FLOAT` to `IEEE` is reflected in the macros that are predefined by default.

Some users have tried defining the macro `__ALPHA` explicitly by using `/DEFINE` or by putting it in a header file as a quick way to deal with source code conditionals that were written to assume that if `__ALPHA` is not defined, then the target must be a VAX. Doing this will cause the CRTL headers and other OpenVMS headers to take the wrong path for I64. You must not define any of the Alpha architecture predefined macros when using this compiler.

6.5 Java

Java Version 1.4.2 is supported on OpenVMS I64. There are no differences between Java on OpenVMS Alpha and OpenVMS I64.

6.6 Macro-32

Most VAX MACRO programs that compile on OpenVMS Alpha should recompile on OpenVMS I64 without modification. However, programs that call routines with nonstandard return values or programs that use the JSB instruction to call routines written in other languages must add some new directives to the source file. For more information, refer to the *HP OpenVMS MACRO Compiler Porting and User's Guide*.

Other Considerations

This chapter describes additional porting considerations.

7.1 Hardware Considerations

This section contains information about the Alpha, PA-RISC and the Itanium processor families. It also discusses processor-related development issues and provides examples using both C and assembly language for typical processor-dependent code.

7.1.1 Itanium Processor Family Overview

The Itanium architecture was developed by HP and Intel as a 64-bit processor for engineering workstations and e-commerce servers. The Itanium processor family uses a new architecture called Explicitly Parallel Instruction Computing (EPIC).

Some of the most important goals in the design of EPIC were instruction-level parallelism, software pipelining, speculation, predication, and large register files. These allow Itanium processors to execute multiple instructions simultaneously. EPIC exposes many of these parallel features to the compiler (hence "Explicitly" in the name). This increases the complexity of the compiler, but it allows the compiler to directly take advantage of these features.

EPIC is designed so future generations of processors can utilize different levels of parallelism, which increases the flexibility of future designs. This allows the machine code to express where the parallelism exists without forcing the processor to conform to previous machine widths.

For more information on assembly programming for the Itanium processor family, see the Intel Itanium Architecture Assembly Language Reference Guide. Available at:

http://developer.intel.com/software/products/opensource/tools1/tol_whte2.htm

7.1.2 Alpha Processor Family Overview

Alpha is a 64-bit load/store RISC architecture that is designed with particular emphasis on the three elements that most affect performance: clock speed, multiple instruction issue, and multiple processors.

The Alpha architects examined and analyzed current and theoretical RISC architecture design elements and developed high-performance alternatives for the Alpha architecture.

The Alpha architecture is designed to avoid bias toward any particular operating system or programming language. Alpha supports the OpenVMS Alpha, TRU64 UNIX, and Linux operating systems and supports simple software migration for applications that run on those operating systems.

Other Considerations

7.1 Hardware Considerations

Alpha was designed as a 64-bit architecture. All registers are 64 bits in length and all operations are performed between 64-bit registers. It is not a 32-bit architecture that was later expanded to 64 bits.

The instructions are very simple. All instructions are 32 bits in length. Memory operations are either loads or stores. All data manipulation is done between registers.

The Alpha architecture facilitates pipelining multiple instances of the same operations because there are no special registers and no condition codes.

The instructions interact with each other only by one instruction writing a register or memory and another instruction reading from the same place. That makes it particularly easy to build implementations that issue multiple instructions every CPU cycle.

Alpha makes it easy to maintain binary compatibility across multiple implementations and easy to maintain full speed on multiple-issue implementations. For example, there are no implementation-specific pipeline timing hazards, no load-delay slots, and no branch-delay slots.

The Alpha architecture reads and writes bytes between registers and memory with the LDBU and STB instructions. (Alpha also supports word read/writes with the LDWU and STW instructions.) Byte shifting and masking is performed with normal 64-bit register-to-register instructions, crafted to keep instruction sequences short.

As viewed from a second processor (including an I/O device), a sequence of reads and writes issued by one processor may be arbitrarily reordered by an implementation. This allows implementations to use multibank caches, bypassed write buffers, write merging, pipelined writes with retry on error, and so forth. If strict ordering between two accesses must be maintained, explicit memory barrier instructions can be inserted in the program.

The basic multiprocessor interlocking primitive is a RISC-style `load_locked`, `modify`, `store_conditional` sequence. If the sequence runs without interrupt, exception, or an interfering write from another processor, then the conditional store succeeds. Otherwise, the store fails and the program eventually must branch back and retry the sequence. This style of interlocking scales well with very fast caches and makes Alpha an especially attractive architecture for building multiple-processor systems.

A Privileged Architecture Library (PALcode) is a set of subroutines that are specific to a particular Alpha operating system implementation. These subroutines provide operating-system primitives for context switching, interrupts, exceptions, and memory management. PALcode is similar to the BIOS libraries that are provided in personal computers.

For more information, please see the *Alpha Architecture Handbook*, part number EC-QD2KC-TE.

7.2 Endianism Considerations

Users who are porting from OpenVMS Alpha or OpenVMS VAX to OpenVMS I64 do not need to be concerned with endianism issues; OpenVMS on all platforms uses little-endian for data storage and manipulation. This chapter is being made available more for reference to allow the reader to be aware of issues, in general, when porting between different hardware architectures where endianisms are different.

Other Considerations

7.2 Endianism Considerations

Endianism refers to the way in which data is stored, and defines how bytes are addressed in multibyte data types. This is important because if you try to read binary data on a machine that is of a different endianism than the machine that wrote the data, your results will be different. This is an especially significant problem when a database needs to be moved to a different endian architecture. There are two types of endian machines: big-endian (forward byte ordering or Most Significant First, MSF) and little-endian (reverse byte ordering or Least Significant First, LSF). The mnemonics of "Big End In" and "Little End In" can be useful when discussing endianism. Figure 7–1 and Figure 7–2 compare the two byte ordering methodologies graphically.

Figure 7–1 Big-Endian Byte Ordering

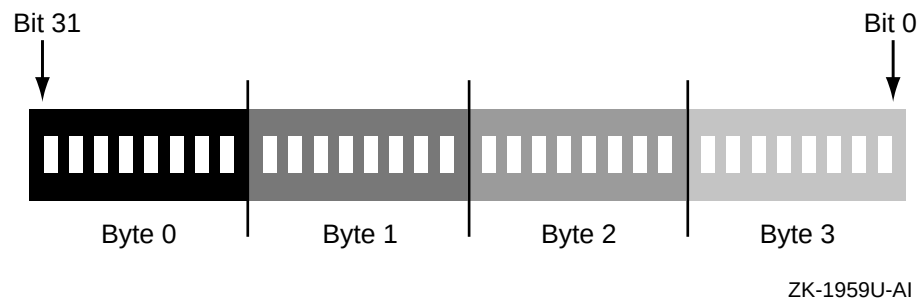
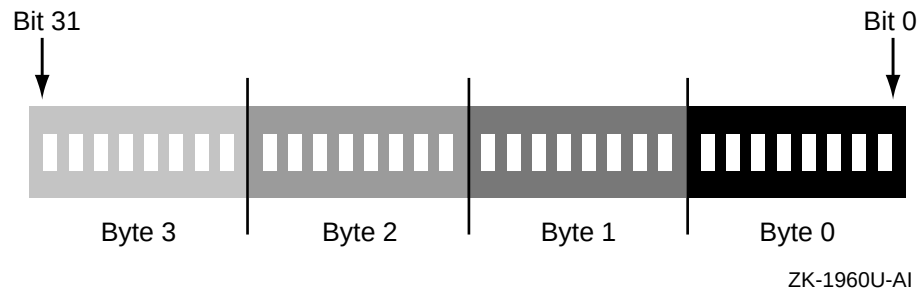


Figure 7–2 Little-Endian Byte Ordering



The Alpha microprocessor provides a little-endian environment for operating systems such as OpenVMS and TRU64 UNIX while the PA-RISC microprocessor provides a big-endian environment for the HP-UX UNIX operating system.

On a little-endian operating system, such as OpenVMS, the little end, or least-significant byte is stored at the lowest address. This means a hexadecimal like 0x1234 is stored in memory as 0x34 0x12. The same is true for a 4-byte value; for example, 0x12345678 is stored as 0x78 0x56 0x34 0x12. A big-endian operating system does this in the reverse fashion, so 0x1234 is stored as 0x12 0x34 in memory.

Other Considerations

7.2 Endianism Considerations

Consider the number 1025 (2 to the tenth power plus one) stored in a 4-byte integer. This is represented in memory as shown in the following example:

```
little endian: 00000000 00000000 00000100 00000001b  
big endian:  00000001 00000100 00000000 00000000b
```

The Itanium processor family supports both big- and little-endian memory addressing. There are no issues reading and exchanging data between OpenVMS Alpha and OpenVMS I64 running on Itanium processor family. If there is a need to exchange data between OpenVMS, on either Alpha or the Itanium processor family, with HP-UX, on either PA-RISC or the Itanium processor family, then byte swapping may be required.

Application Evaluation Checklist

Use this checklist to guide you through the planning and porting process.

Application Evaluation Checklist

Application Evaluation Checklist

General Information

1. What does your application do (brief description)?

Development History and Plans

2. a. Does the application currently run on other operating systems or hardware architectures? ☐ YES ☐ NO
- b. If yes, does the application currently run on the latest OpenVMS Alpha system? ☐ YES ☐ NO

3. When was the last time the application environment was completely recompiled and rebuilt from source? _____

- a. Is the application rebuilt regularly? ☐ YES ☐ NO
- b. How frequently? _____

4. Is the application actively maintained by developers who know it well? ☐ YES ☐ NO

List developer names and contact information:

Developer	Contact
_____	_____
_____	_____
_____	_____

5. a. How is a new build of the application tested or verified for proper operation?

- b. Do you have performance characterization tools to assist with optimization? ☐ YES ☐ NO

- c. If yes, list the tools and version numbers used:

Tool	Version
_____	_____
_____	_____
_____	_____

- d. Which source-code configuration management tools are used? List the tools and their version numbers:

Tool	Version
_____	_____
_____	_____

Application Evaluation Checklist

- _____
6. Do you have a development and test environment separate from your production systems? ☐ YES ☐ NO
7. What procedures are in place for rolling in a new version of the application into production?

8. What are your plans for the application after porting it to OpenVMS I64?
- | | | |
|--|------------------------------|-----------------------------|
| a. No further development | ☐ YES | ☐ NO |
| b. Maintenance releases only | ☐ YES | ☐ NO |
| c. Additional or changed functionality | ☐ YES | ☐ NO |
| d. Maintain separate OpenVMS I64 and OpenVMS Alpha sources | ☐ YES | ☐ NO |

Composition of the Application

9. How large is your application?
How many modules? _____
How many lines, disk blocks, or megabytes of code? _____
How much disk space is required? _____
10. a. Do you have access to all source files that make up your application? ☐ YES ☐ NO
b. If you are considering using HP Services, will it be possible to give HP access to these source files and build procedures? ☐ YES ☐ NO
11. a. List the languages used to write the application. (If multiple languages are used, give the percentages of each.)
- | Language | Percentage |
|----------|------------|
| _____ | _____ |
| _____ | _____ |
| _____ | _____ |
- b. Do you use Macro-64, PL/I, ADA 83, or Fortran 77 on OpenVMS Alpha? ☐ YES ☐ NO
12. Is there application documentation? ☐ YES ☐ NO

External Dependencies

13. What is the system configuration (CPUs, memory, disks) required to set up a development environment for the application?
- _____

Application Evaluation Checklist

14. What is the system configuration (CPUs, memory, disks) required to set up a typical user environment for the application, including installation verification procedures, regression tests, benchmarks, or workloads?

15. Does the application rely on any special hardware? ☐ YES ☐ NO

16. What version of OpenVMS Alpha does your application currently run on?

Does the application run on OpenVMS Alpha Version 7.3-2 or 7.3-1? ☐ YES ☐ NO

17. Does the application require layered and third-party products to run?

a. From HP: (other than compiler RTLs) ☐ YES ☐ NO

List the HP layered products and versions:

HP Product	Version

b. From third parties: ☐ YES ☐ NO

List the third-party products and versions:

Third-Party Product	Version

18. a. Do you have regression tests for the application? ☐ YES ☐ NO

b. If yes, do they require any particular software product such as HP's Digital Test Manager? ☐ YES ☐ NO

Dependencies on the OpenVMS Alpha Architecture

19. Does the application use OpenVMS VAX floating data types? ☐ YES ☐ NO

20. Is the data naturally aligned? ☐ YES ☐ NO

21. Does the application call OpenVMS system services that:

a. Modify the working set? ☐ YES ☐ NO

b. Change program control flow using SYS\$GOTO_UNWIND? ☐ YES ☐ NO

Application Evaluation Checklist

22. a. Does the application use multiple, cooperating processes? ☐ YES ☐ NO
 If so:
 b. How many processes? _____
 c. What interprocess communication method is used? _____
- ☐ \$CRMPSC ☐ Mailboxes ☐ SCS ☐ Other
☐ DLM ☐ SHM, IPC ☐ SMG\$ ☐ STR\$
- d. If you use global sections (\$CRMPSC) to share data with other processes, how is data access synchronized? _____
23. Does the application currently run in a multiprocessor (SMP) environment? ☐ YES ☐ NO
24. Does the application use AST (asynchronous system trap) mechanisms? ☐ YES ☐ NO
25. a. Does the application contain condition handlers? ☐ YES ☐ NO
 b. Do any handlers modify data in the mechanism array? ☐ YES ☐ NO
26. a. Does the application run in privileged mode or link against SYS\$BASE_IMAGE? ☐ YES ☐ NO
 If so, why? _____
 b. Does the application depend on OpenVMS internal data structures or interfaces? ☐ YES ☐ NO
27. Does the application use connect-to-interrupt mechanisms? ☐ YES ☐ NO
 If yes, with what functionality? _____
28. Does the application create or modify machine instructions? ☐ YES ☐ NO
29. Does the application include any other user-written code or assembler code that depends specifically on the OpenVMS Alpha architecture? ☐ YES ☐ NO
30. Does the application include or depend on any conditionalized statements in source code or build files, or any code that includes logic that assumes it is running on either a VAX or an OpenVMS Alpha system? ☐ YES ☐ NO
31. What parts of the application are most sensitive to performance? I/O, floating point, memory, realtime (that is, interrupt latency, and so on).

Application Evaluation Checklist

- | | | | |
|-----|---|------------------------------|-----------------------------|
| 32. | Does the application use any OpenVMS Alpha calling standard routines? | ☐ YES | ☐ NO |
| 33. | Does the application use non-standard routine linkage declarations? | ☐ YES | ☐ NO |
| 34. | a. Does the application depend on the format or content of an object file? | ☐ YES | ☐ NO |
| | b. Does the application depend on the format or content of an executable image? | ☐ YES | ☐ NO |
| | c. Does the application depend on the debug symbol table content of an image? | ☐ YES | ☐ NO |
| 35. | Does the application use the C asm statement? | ☐ YES | ☐ NO |
| 36. | Does the application use BLISS register BUILTINS? | ☐ YES | ☐ NO |

Porting Your Application

- | | | | |
|-----|--|------------------------------|-----------------------------|
| 37. | Have you upgraded your OpenVMS Alpha system to the latest version currently available? | ☐ YES | ☐ NO |
| 38. | a. Have you test compiled your application on the latest version of OpenVMS Alpha, using the most recent version of the compiler? | ☐ YES | ☐ NO |
| | b. Have you corrected any problems detected during the compile? | ☐ YES | ☐ NO |
| 39. | Copy the application source modules and all related code to the OpenVMS I64 system. Have you copied the modules and related code to OpenVMS I64? | ☐ YES | ☐ NO |
| 40. | Compile, link, and run the application on the OpenVMS I64 system.
Application is compiled, linked, and running? | ☐ YES | ☐ NO |
| 41. | Perform unit testing on the OpenVMS I64 system.
Have you completed the unit testing? | ☐ YES | ☐ NO |
| 42. | Perform system testing on the OpenVMS I64 system.
Have you completed the system testing? | ☐ YES | ☐ NO |
| 43. | Perform regression testing on the OpenVMS I64 system.
Have you completed the regression testing? | ☐ YES | ☐ NO |
| 44. | Did testing succeed? | ☐ YES | ☐ NO |

Glossary

ABI

Application Binary Interface. A general term encompassing all of the rules and conventions that programs must follow to run under a particular operating system. For example, the OpenVMS I64 ABI includes the calling standard, register usage, interrupt handling, and other related topics.

According to Intel's *Itanium Software Conventions and Run-Time Architecture Guide*, "an ABI is composed of an API, system-specific conventions, a hardware description, and a run-time architecture."

ACPI

Advanced Configuration and Power Interface. ACPI provides interfaces for power state management, system device identification, and hardware configuration.

- ACPI specification:
<http://www.acpi.info/spec.htm>
- Intel's ACPI website:
<http://developer.intel.com/technology/iapc/acpi/>

ALAT

Advanced Load Address Table. A lookup table on Itanium that tracks **speculative** memory accesses and allows the CPU to handle conflicts.

AML

ACPI Machine Language. Pseudocode for a virtual machine supported by an ACPI-compatible operating system and in which ACPI control methods and objects are written. See also **ACPI**.

application registers

128 special-purpose registers used for various functions, labeled ar0 to ar127. For example, ar17 is the register backing store pointer.

Aries

HP emulator/translator for HP Precision Architecture (**PA**) application code. Unlike **VEST** (the VAX to Alpha translator), Aries emulates only at run-time and does not produce a translated image. When the image exits, all the translated code is discarded and must be regenerated each time the program runs.

ASL

ACPI Source Language. The programming language for **AML**. ASL source code is compiled into AML images.

ASM

Address Space Match, part of virtual memory management.

branch registers

Itanium 64-bit registers, br0 to br7, used to specify the target addresses for indirect branches. The branch registers streamline call/return branching.

bundle

The basic Itanium instruction format. A bundle is a 128-bit structure consisting of three 41-bit instructions, plus a 5-bit **template**.

CISC

Complex Instruction Set Computing. Includes numerous variable-length machine instructions that are more complex than RISC. See also **RISC** and **VLIW**.

COW

Copy-on-write. Refers to memory that is mapped by multiple processes provided each process does not attempt to write to it. If a process writes to a page, the page is copied to a process-private location and remapped, leaving the original shared page intact. COW is commonly used by the loader for memory-mapped shared libraries.

CS

Calling Standard

CVF

Compaq Visual Fortran

DSDT

Differentiated System Description Table. Part of the **ACPI** subsystem. The DSDT contains the Differentiated Definition Block, which supplies the implementation and configuration information about the base system.

DVD

Digital Versatile Disk.

DWARF

Debugging and traceback information (embedded in **ELF**).

EFI

Extensible Firmware Interface. EFI's purpose is to initialize the hardware and start booting an operating system.

For more information, see Intel's EFI Page:

<http://developer.intel.com/technology/efi/index.htm>

ELF

Executable and Linkable Format (ELF) defines the file format for object files, shared libraries, and executable images on Itanium. ELF is not specific to Itanium; it has been used for years on various UNIX systems.

For more information, see the ELF specification from SCO:

<http://stage.caldera.com/developer/gabi/>

EPIC

Explicitly Parallel Instruction Computing. A term coined by Intel to describe the Itanium architecture, as compared to **CISC** or **RISC**. EPIC provides architecturally visible access to parallelism in the CPU. For example, the Itanium Processor Family defines data speculation and register renaming as part of the architecture, rather than hiding it in the chip implementation as was done with Alpha. EPIC is similar to the Very Long Instruction Word (**VLIW**) computing architecture.

FADT

Fixed ACPI Description Table. A table that contains the **ACPI** Hardware Register Block implementation and configuration details, as well as the physical address of the **DSDT**.

FAT

File Allocation Table. The disk volume structure used by various Microsoft operating systems and by the Itanium **EFI** operating system loader. There are various FAT variants, including FAT8, FAT12, FAT16, and FAT32. These differ in total permitted volume size and various other minor or random changes.

fills

See **spills and fills**.

FIT

Firmware Interface Table

GEM

The backend code generator used by most HP Alpha and Itanium compilers.

general registers

Itanium has 128 64-bit general registers, numbered as r0 to r127 (sometimes called gr0 to gr127). However, only 32 registers are normally used directly; the other 96 are used for argument passing and to store local variables for the currently executing procedure. Registers r0 to r31 are static, but registers r32 and up are generally managed by the Register Stack Engine (**RSE**) and might be renamed across procedure calls.

global pointer (GP)

The base address of the current global data segment. The base address allows the use of compact, GP-relative addresses in machine code. Because the largest immediate value in an Itanium instruction is 22 bits wide, the global data segment is 4 MB in length. General register 1 holds the current GP.

HWPCB

Hardware Process Control Block. The portion of the process context that is stored and retrieved by the hardware when a process is rescheduled.

HWRPB

Hardware Restart Parameter Block. These data structures are used to communicate system-level configuration information between the console and OpenVMS both on system startups and system restarts.

IA

Intel Architecture

IA-32

An Intel 32-bit CISC microprocessor architecture. Examples of implementations include the Pentium, Xeon, and 80386 series.

IA-64

An old name for the Itanium Processor Family architecture. The term IA-64 is no longer officially used. However, it still appears in older documentation, and IA64 still occurs in some code.

Integrity

The brand name of the HP Itanium server product line. For example, the rx2600 server is sold as the "HP Integrity rx2600".

ILP

Instruction Level Parallelism. The ability to execute multiple instructions in parallel during the same cycle.

instruction group

A sequence of one or more instructions, delimited by explicit **stops** or taken branches. Instructions within an instruction group must have no **RAW** or **WAW** dependencies. Instruction groups can cross **bundle** boundaries.

IPINT

Inter-Processor Interrupt. Also simply called IPI.

IPF

Itanium Processor Family

IPMI

Intelligent Platform Management Interface, a set of APIs used to manage servers. Functions include monitoring environmental data such as temperature, voltage, fans, and chassis switches, as well as alerting, remote console and control, and asset tracking. See Intel's website for more details:

<http://www.intel.com/design/servers/ipmi/>

Itanium

The name of both the 64-bit architecture (the Itanium Processor Family) and the first chip to implement the architecture (the Itanium Processor).

Itanium 2

The second generation in the Itanium Processor Family. The **rx2600** is an Itanium-2 system. For more information, refer to the following website:

http://www.cpus.hp.com/technical_references/ia64.shtml

IVT

Interrupt Vector Table

MAS

Multiple Address Space. In a MAS operating system, all processes on the system have at least a partially private virtual address space. OpenVMS and Tru64 UNIX are both MAS operating systems. See also **SAS**.

MBR

Master Boot Record. The contents of the first disk block and the core data structure of a FAT32 partitionable disk.

NaT

Not a Thing. An extra bit associated with various registers that indicates whether or not the register contents are valid. Used for **speculative** execution. For example, a read from a nonexistent virtual address would result in NaT. The NaT bit propagates as execution progresses so that any operation involving a NaT results in a NaT.

NMI

Non-Maskable Interrupt

NUE

The Linux Native User Environment, a set of development software used in conjunction with the **Ski** Itanium emulator.

PA

Hewlett Packard (HP) Precision Architecture, a computer architecture also known as PA-RISC. HP is porting its UNIX from PA to Itanium.

PAL (Alpha)

Privileged Architecture Library. PAL is part of the Alpha architecture. It is a software mechanism for performing low-level operations such as interrupt handling, **TLB** management, and atomic operations that were implemented on VAX as microcode. On Itanium, the same functionality is provided as part of the operating system. Whenever possible, the OpenVMS Itanium Bliss, C, and Macro compilers convert CALL_PAL macros to the equivalent operating system calls for backward compatibility. Not all Alpha PAL operations will be implemented on IPF; in some cases, programs calling PALcode directly may need to change.

PAL (Itanium)

Processor Abstraction Layer. Part of the Itanium architecture that is implemented in firmware. Provides a consistent interface for processor-specific functions such as hardware errors or system initialization. Generally speaking, **SAL** isolates the operating system from platform-specific implementation differences, while PAL isolates the operating system (and SAL) from processor-specific differences.

PIC

Position-independent code. Machine code that uses only relative address references. This strategy allows the code to be loaded to any memory location without modification.

POSSE

Pre-Operating System Startup Environment. POSSE will be the standard firmware user interface on **Itanium 2** servers. It is an HP value-added component that is available only on HP systems.

PPN

Physical Page Number

PTE

Page Table Entry

predication

The conditional execution of instructions based on a set of special 1-bit registers called predicates. Most Itanium instructions include a predicate number. When the corresponding predicate register is true (1), the instruction is executed. When it is false (0), the instruction is treated as a "no-op".

RAW

Read-after-write violation. A type of data dependency between two instructions in one instruction group. The later instruction reads data from the same register to which the earlier instruction wrote.

RID

Region Identifier

RISC

Reduced Instruction Set Computing. Fixed-size machine instructions that are intended to be simpler machine instructions than CISC. See also **CISC** and **VLIW**.

RSE

Register Stack Engine. An on-chip entity that handles register renaming and **spills and fills** across procedure calls.

rx2600

An HP server with up to two Itanium-2 CPUs. The rx2600 is the first platform for customer use to run the OpenVMS Itanium port.

SAL

System Abstraction Layer. Part of the Itanium architecture, implemented in firmware. Provides a consistent interface for platform-specific functions such as hardware errors or system initialization. Generally speaking, SAL isolates the operating system from *platform*-specific implementation differences, while **PAL** isolates SAL and the operating system from *processor*-specific differences.

SAS

Single Address Space. In a SAS operating system, all processes share the same address space. See also **MAS**.

SCI

System Control Interrupt. A system interrupt used by hardware to notify the operating system about **ACPI** events.

Ski

A free Itanium emulator for Linux. Ski emulates the Itanium Processor Family architecture, but not any particular implementation.

speculation

The process of moving memory accesses earlier in the code to hide memory access latencies. If it turns out that the memory access was invalid or unnecessary, it can easily be discarded. The Itanium `ld.a`, `ld.s`, `chk.a`, and `chk.s` instructions are used for speculation.

spills and fills

The rotating portion of the Itanium register file that is used as a stack. If it overflows, registers are saved to a memory region called the register backing store. A *spill* flushes one or more registers out to the backing store; a *fill* restores registers from the backing store. Spills and fills are handled automatically by the Register Stack Engine (**RSE**) on the CPU without program intervention.

SRM console

Firmware for Alpha systems that includes boot support, PALcode, and platform-specific functionality used by OpenVMS, UNIX, and Linux. The chevron prompt (`>>>`) that you see when you power on any Alpha system is part of the SRM console. The console behavior is defined in the *Alpha System Reference Manual* (SRM).

The Itanium architecture does not have an equivalent to the SRM console. Instead, the SRM functionality is split between the **PAL/SAL** firmware, the **EFI** boot software, and the operating system.

stop

The end of an **instruction group**, designated in by a double semicolon (`::`) in assembly listings.

Superdome

The high-end HP server product line. Superdomes support about 64 CPUs, either PA-RISC or Itanium. Eventually, all new Superdome systems will be Itanium based.

SWIS

SoftWare Interrupt Support. The set of services that implement the interrupt model used by OpenVMS. This includes processor IPL, ASTs, mode changes, and software interrupts. On Alpha this support was implemented in PALcode.

template

Templates define the combinations of instructions that can be placed into a single Itanium **bundle**. For example, a bundle with the MFI template must have, in this order, a memory instruction (M), a floating-point instruction (F), and an integer instruction (I). Templates also define the location of **stops**. 24 templates are defined in the Itanium architecture, but there are only 12 combinations of letters. For each combination, there are two template versions: one with a stop at the end, and one without a stop. Some templates also contain stops within the bundle.

TLB

Translation Lookaside Buffer. An on-chip cache that stores recently used mappings between real and virtual memory. Itanium has separate TLBs for instructions and data.

USB

Universal Serial Bus. A connector for I/O devices such as keyboards, mice, printers, cameras, external disks, and so forth.

VEST

A tool that translates VAX executable images to Alpha executable images. Also known as DECMigrate.

VHPT

Virtual Hash Page Table

VIAL

VMS to Itanium Abstraction Layer. VIAL is the equivalent of the OpenVMS SRM console PALcode layer.

VLIW

Very Long Instruction Word computing architecture. Exceedingly complex instructions reminiscent of the microcode engine found deep within many VAX processors. See also **CISC**, **RISC**, and **EPIC**.

VPN

Virtual Page Number

VRN

Virtual Region Number

WAW

Write-after-write violation. A type of data dependency between two instructions in one instruction group. Both of the instructions write to the same destination register.

WWID

World-Wide ID. A globally-unique Fibre Channel Device Identifier analogous to an Ethernet station address.

A

- ADA 83, 3–11, A–3
- Analyze/Image utility (ANALYZE/IMAGE), 5–2
- Analyze/Object utility (ANALYZE/OBJECT), 5–2
- Application
 - size, 3–10, A–3
- Applications
 - and compilers, 3–11
 - architecture dependencies, 3–13, A–4
 - assessing, 3–2
 - contents being ported, 3–3
 - dependencies, 3–5, 3–12, A–3
 - operational tasks, 3–7
 - porting method, 3–2
 - calling standard, 3–3, 3–15, A–6
 - components, 3–10, A–3
 - development environment, 3–6
 - documentation, 3–11, A–3
 - establishing baseline values for, 4–4
 - hardware dependencies, 3–12, A–4
 - languages used, 3–11, A–3
 - machine instructions, 3–15, A–5
 - operating environment, 3–6
 - performance sensitivity, 3–15, A–5
 - requiring changes, 3–3
 - size, 3–10, A–3
 - software dependencies, 3–5, 3–12, A–4
 - synchronization among parts of, 3–14, A–5
 - use of global sections, 3–14, A–5
- Architecture dependencies, 3–13, A–4
- asm statement, 3–16
- ASTs (asynchronous system traps), 3–14, A–5

B

- Baseline values for application
 - establishing, 4–4
- Binary translator, 3–3, 4–1, 5–3
- BLISS BUILTIN, 3–16, A–6
- BLISS compilers
 - differences between Alpha and I64, 6–2
- Bugs
 - latent, 4–6

C

- Calling standard, 3–3, 3–15, A–6
 - code that relies on, 4–15
- C asm statement, 3–16, A–6
- CLUE (Crash Log Utility Extractor)
 - See Crash Log Utility Extractor
- CMA_DELAY API library routine, 4–14
- CMA_TIME_GET_EXPIRATION API library routine, 4–14
- CMS (Code Management System), 4–3
- COBOL, 6–15
- Code
 - assembler, 3–15, A–5
 - requiring changes, 3–3
 - user-written, 3–15, A–5
- Compile commands
 - changes required, 4–4
- Compile procedures, 4–3
- Compilers
 - assessing porting support, 3–11
 - commands, 4–4
 - native Alpha, 5–1
 - native I64, 5–1
 - optimizing, 5–1
 - preparing to port applications, 6–1
 - qualifiers for VAX floating point data types, 5–1
- Conditionalized code, 3–3, 3–15, 4–7, A–5
- Condition handlers, 3–14, 4–10, A–5
- Connect-to-interrupt mechanisms, 3–15, A–5
- Crashes
 - analyzing, 4–5
- Crash Log Utility Extractor (CLUE), 4–5, 5–3
- \$CRMPSC, 3–14, A–5

D

- Data alignment, 3–5, 3–13, 4–14 to 4–15, A–4
 - compiler options, 4–14
 - finding unaligned data, 4–14
 - incompatibility with translated software, 4–15
 - performance, 4–14
- Data structures
 - alignment, 3–5, 3–13, A–4
 - floating-point, 3–5, 3–7, 3–13, A–4

Data structures (cont'd)

- internal, 3-3, 3-14, A-5
- mechanism array, 3-5, 3-14, A-5

Data types

- floating-point, 3-5, 3-7, 3-13, A-4

Debuggers, 4-4

- Delta, 5-2
- native I64, 5-2
- XDelta, 5-2

Debugging, 5-2

Debug symbol table, 4-10

Debug symbol table format, 3-3, 3-16, A-6

DECset, 5-2

Delta Debugger, 5-2

Digital Test Manager, 3-13, A-4

Drivers

- terminal, 3-5

Dump files

- See System dump files

DWARF image file format, 4-10

E

ELF object file format, 4-10

Executable image file format, 3-3, 3-16, A-6

F

Floating-point data types, 3-5, 3-7, 3-13, A-4

- IEEE, 4-11
- VAX, 4-11, 5-1

Fortran, 6-15

Fortran 77, 3-11, A-3

H

HP C, 6-17

HP rx2600 system, 4-2

HP Services support, 3-8

I

Include files

- for C programs, 4-3

Internal data structures, 3-3, 3-14, A-5

Interoperability

- confirming, 4-6

J

Java, 6-18

Jump to Subroutine (JSB), 3-5

L

Languages, 3-11, A-3

LIB\$LOCK_IMAGE routine, 4-16

LIB\$UNLOCK_IMAGE routine, 4-16

LIB\$WAIT

- common code for I64 and Alpha, 4-12

Librarian utility (LIBRARIAN)

- native Alpha, 5-2

Library interfaces, 3-1, 3-3

Linkage declarations, 3-16, A-6

Link commands

- changes required, 4-4

Linker utility

- native I64, 5-2

Link procedures, 4-3

Logical names

- for tools and files, 4-3

M

Machine instructions

- correct execution of, 3-15, A-5

Macro-32 compiler, 6-18

MACRO-32 compiler, 5-1

Macro-64, 3-11, A-3

MACRO code

- replacing, 3-11, A-3

Mechanism array data structure, 3-5, 4-10

Message utility (MESSAGE)

- native Alpha, 5-2

Migration environment

- setting up, 4-2

Migration tools, 4-3

MMS (Module Management System), 4-3

Multiprocessing, 3-14, A-5

MULTITHREAD system parameter, 4-13

O

Object file format, 3-3, 3-16, A-6

- reliance on, 4-10

P

PGFIPLHI bugchecks, 4-15

PL/1, 3-11, A-3

Porting applications, 6-1

Porting checklist, 3-8, 7-4

Porting dependencies

- external, 3-5, 3-12, A-3

Porting method, 3-2

Porting needs

- assessing, 3-1

Porting resources, 3-8

- Porting steps, 3–16, A–6
- Privileged mode operation, 3–14, A–5
- Programming languages
 - BASIC, 5–1
 - BLISS, 5–1
 - C, 5–1
 - include files, 4–3
 - C++, 5–1
 - COBOL, 5–1
 - Fortran, 5–1
 - Java, 5–1
 - Pascal, 5–1
 - VAX MACRO, 5–1
- Program sections
 - overlaid, 4–10

R

- Recompiling
 - changes in compile commands, 4–4
 - resolving errors, 4–4
- Relinking, 5–2
 - changes in link commands, 4–4

S

- SDA (System Dump Analyzer utility)
 - See System Dump Analyzer utility
- SS\$_FLTDIV exception, 4–10
- SS\$_FLTINV exception, 4–10
- SS\$_HPARITH exception, 4–10
- Synchronization
 - of interprocess communication, 3–14, A–4
- SYS\$BASE_IMAGE
 - linking against, 3–14, A–5
- SYS\$GOTO_UNWIND system service, 3–13, 4–9, A–4
- SYS\$GOTO_UNWIND_64 system service, 4–9
- SYS\$LKWSET system service, 3–13, 4–15, A–4
- SYS\$LKWSET_64 system service, 4–15
- System-Code Debugger, 4–4
 - See also Debugger
- System Dump Analyzer utility (SDA), 5–2
 - OpenVMS Alpha, 4–5
- System dump files
 - analyzing, 4–5
- System services, 3–1, 3–3
 - SYS\$GOTO_UNWIND, 3–13, 4–9, A–4

- SYS\$GOTO_UNWIND_64, 4–9
- SYS\$LKWSET, 3–13, 4–15, A–4
- SYS\$LKWSET_64, 4–15

T

- Terminal drivers, 3–5
- Testing applications
 - compile, 3–17, A–6
 - establishing baseline values on Alpha systems, 4–4
 - on I64 systems, 4–5
 - regression testing, 3–17, A–6
 - regression tests, 3–13, A–4
 - system testing, 3–17, A–6
 - unit testing, 3–17, A–6
- Test tools, 4–3
 - I64 specific, 4–6
 - ported to I64, 4–5
- THREADCP command, 4–13
- Threading-like code, 3–5
- Thread interfaces
 - legacy API library routines, 4–14
 - support on I64, 4–13
- Threads
 - multiple kernel, 4–13
- Translating, 3–2, 3–10, 3–11, 5–3, A–3
 - programs in languages with no I64 compiler, 5–3
- Translator
 - See Binary translator

V

- VAX MACRO
 - recompiling on OpenVMS I64 systems, 5–1

W

- Working set
 - modifying, 3–13, A–4

X

- XDelta Debugger, 4–4, 5–2

