VMS Software

VSI OpenVMS

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

Operating System and Version: VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

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

Legal Notice

Table of Contents

Preface

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

Porting Applications from VSI OpenVMS Alpha to VSI 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 VSI OpenVMS Industry Standard 64 for Integrity Servers (I64) operating system.

3. Document Structure

This document is organized as follows:

Chapter 1 includes an overview of the OpenVMS I64 8.2 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.

Appendix B lists the VSI layered products that are supported on OpenVMS Alpha but not on OpenVMS I64.

Appendix C describes how to use KP routines when porting application-specific stack-switching code to I64.

4. Related Documents

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

- *VSI OpenVMS Calling Standard*
- *VSI OpenVMS Debugger Manual*

- *VSI OpenVMS Linker Utility Manual*
- *VSI OpenVMS System Analysis Tools Manual*

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

5. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

6. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

7. Typographical Conventions

The following conventions may be used in this manual:

Convention	Meaning
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.
. . .	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.

Convention	Meaning
{ }	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.
italic type	Italic type indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error*number*), in command lines (/PRODUCER= *name*), and in command parameters in text (where *dd* 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 radixes—binary, octal, or hexadecimal—are explicitly indicated.

Chapter 1. Introduction

This chapter contains an introduction to OpenVMS on Industry Standard. This chapter contains an introduction to OpenVMS Industry Standard 64 for Integrity Servers (OpenVMS I64) systems and an overview of the migration process.

For more information, see the *VSI OpenVMS Release Notes*.

1.1. OpenVMS Industry Standard 64 for Integrity Servers

The 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 OpenVMS I64 architecture with few changes ported to the OpenVMS I64 architecture with few changes.

The OpenVMS Alpha and 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, thereby minimizing the time required to perform qualification testing, and helping to ensure the availability of critical applications on both OpenVMS platforms.

VSI 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 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 I64. VSI intends to maintain a strict policy of ensuring forward source compatibility so that “well-behaved” applications that currently run on recommended versions of OpenVMS Alpha run successfully on 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 I64.

I64 has the same look and feel that OpenVMS customers are familiar with. Minor changes were needed to accommodate the new architecture, I64 has the same look and feel familiar to OpenVMS customers. Minor changes were made 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. This section provides an overview of the porting process as it applies to most applications.

1.2.1. Evaluating the Application

Evaluating the application identifies the steps necessary for its migration to VSI 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 requires

- Whether you require support from VSI services

Evaluation should include the following three stages:

1. A general inventory of the application, including identifying dependencies on other software
2. Source analysis to identify dependencies on other software
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; such as:
    - Run-time libraries
    - VSI 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 VSI tools such as Code Management System (CMS) and Module Management System (MMS).

      - Test Suite

        Your tests must be able both to confirm that the migrated application runs correctly and to evaluate its performance.

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

- 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
  - VSI layered products
    - Compilers and compiler RTLs
    - Database managers
    - Networking environment
    - Third-party products

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

    You are 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.

Chapter 2. Fundamental Differences to Consider

This chapter discusses the fundamental differences between the Alpha and OpenVMS 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 started with Itanium conventions and made 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 the *VSI OpenVMS Calling Standard*. For more detailed information, see the *VSI OpenVMS Calling Standard*.

2.1.1. Changes to the OpenVMS Calling Standard

`Data Terminology`–Table 2.1 describes the major changes to the OpenVMS Calling Standard to for I64.

Table 2.1. OpenVMS Calling Standard Changes

Item	Description
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. I64 preserves this characteristic, while the Itanium architecture 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 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.

Item	Description
	OpenVMS uses the same (default) conventions for these registers except in 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.
Parameter passing	OpenVMS parameter passing is similar to the Itanium conventions. Note the following differences in the OpenVMS standard: ● The OpenVMS standards adds 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-point register (F17) slot, and so on. Floating-point parameters are not packed into the available floating-point registers, and a maximum of 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 OpenVMS Calling Standard

`Translated Images` – OpenVMS adds support (signature information and special ABIs) for calls between native and translated VAX or Alpha images. Table 2.2 describes additions to or extensions of the *VSI OpenVMS Calling Standard*.

Table 2.2. OpenVMS Calling Standard Extensions

Item	Description
Floating-point data types	The OpenVMS calling standard 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 80-bit double-extended floating-point type of the Itanium architecture.
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 unwind information block of the Itanium architecture. 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. Floating-Point Arithmetic

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

2.2.1. 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 I64 in one of two ways: continue to use VAX floating-point formats utilizing the conversion features of the VSI 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. VSI compilers generate the code necessary to convert the data between VAX and IEEE formats.

For details about the conversion process, see the *OpenVMS Floating-Point Arithmetic on the Intel® Itanium ® Architecture* white paper.

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.

2.2.2. 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. VSI 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. For detailed descriptions of floating-point options, refer to the individual compilers' documentation.

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

2.3. Object File Format

VSI 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 Itanium 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 can be ported to OpenVMS more easily in the future. All of the compilers, development tools, and utilities provided with the I64 operating system 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.

Chapter 3. Assessing Needs for Porting Your Application

Typically, porting an application to a new platform involves the following procedure:

1. Assessing porting needs
2. Identifying architecture dependencies and nonstandard 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 steps 1 through 4, as necessary
6. Shipping the revised application
7. Operating and installing processes or procedures
8. Supporting and maintaining the application

Although you complete these tasks 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 I64 environment and provides the information necessary to help developers evaluate needs, plan, and begin porting software.

Appendix A provides a checklist to guide you through the planning and porting process.

3.1. Overview

Most applications running on OpenVMS Alpha today can easily be ported to I64 with few changes. The Alpha and OpenVMS 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.

VSI 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 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 I64. However, some published system and library interfaces available on OpenVMS Alpha will not be available or will behave differently on 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.

I64 has the same look and feel familiar to OpenVMS customers. 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 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 does porting require?
- Do you require support from VSI services?

The evaluation process has the following four steps:

1. Identifying applications to be ported
2. Identify dependencies on other software
3. Selecting a porting method
4. Analyzing operational tasks

Completing these steps yields the information necessary to write an effective porting plan. Section 3.2.1 through Section 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.

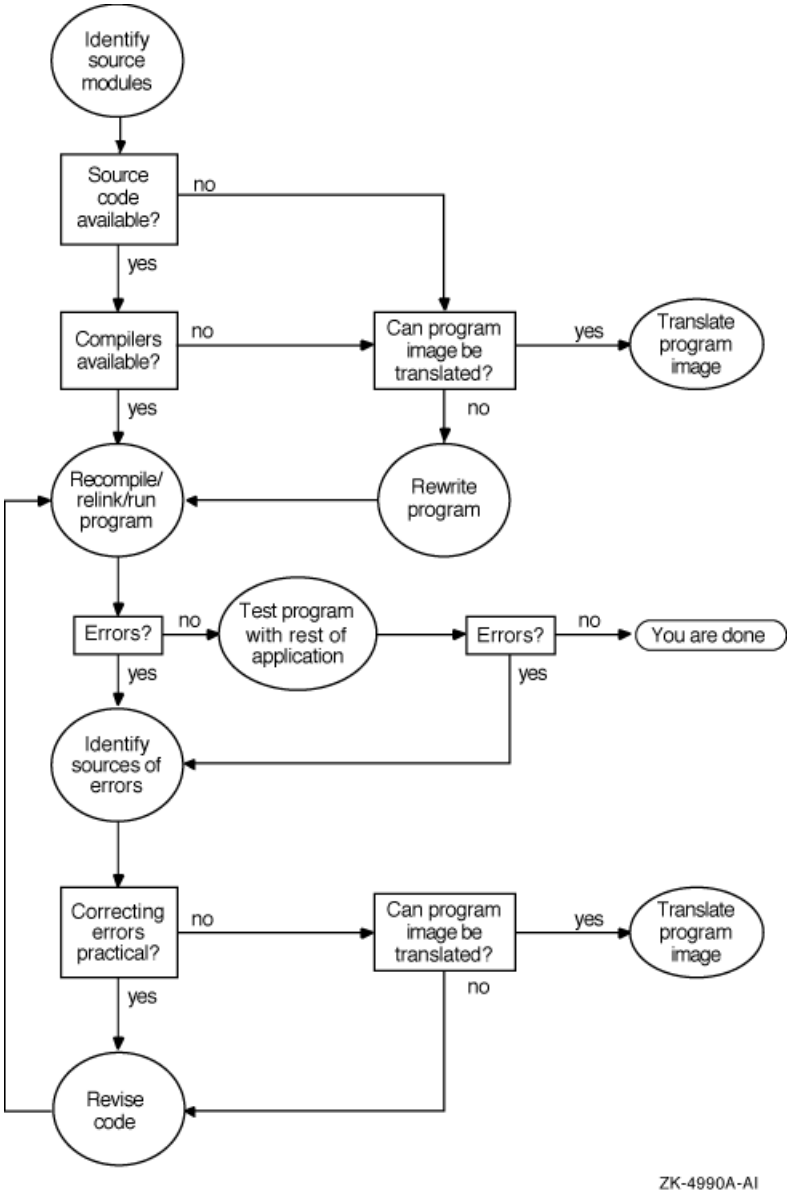
After you complete the evaluation steps described in the following sections, compare your results with the supported product, procedures, and functionality of the target platform. Research any deviations for schedule mismatches, missing functionality, procedural conflicts, and supportability. To complete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

3.2.1. Selecting the Porting Method

After you complete the evaluation process described in the following sections, 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 need to know which methods are possible for each part of the application, and how much work is required for each method. To help answer those questions, you should answer the series of questions and perform the tasks 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.

Figure 3.1. Porting an Application



3.2.2. Identifying Applications to Be Ported

The first step in evaluating an application for porting is to identify 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 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 must be changed. For example, an application may 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 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 I64. (For more information about the availability and divergent behavior of certain system and library interfaces, see Chapter 4 and Chapter 5.)
- 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 which might not be able to handle a third platform.
- 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 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 I64 differs significantly from that on OpenVMS Alpha (see Section 4.8.3.3).

    Note

    VSI recommends that you align your data naturally to achieve optimal performance for data referencing. On both OpenVMS Alpha and 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 4.8.7 and the *VSI 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
- Contains user-written threading-like, such as code that performs direct stack switching, that implements a co-routine or tasking model, or that depends on the nature of the procedure call stack frames. For more details, see Section 4.8.6.

- Incorporates nonstandard or undocumented code practices or interfaces.

In addition any source code that does not have a supported compiler on I64 must 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 on I64; for more information, see the *HP OpenVMS Version 8.2 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 on which your application depends.

3.2.3.1. Software Dependencies

Software dependencies might include such components as:

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

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

- VSI 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
- VSI layered products, including:
  - Compilers and compiler RTLs
  - Database managers
  - Networking environment
- Third-party products

  Many third-party products now run on 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 software 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?

- Test suite

  Your tests should confirm that the ported application runs correctly, and they should evaluate its performance. Regression testing is crucial to the development of any new product, especially to the porting of an application to a new system. 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)
  - Digital Test Manager or an equivalent Open Source tool

3.2.4. Operational Tasks

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

- Installation requirements
- Compilation requirements
  - Is the compiler that you use for your application also supported on this version of I64?
  - Prior to porting your application, VSI 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 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 this version of 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 I64 system, recompile, relink, and requalify. Plan for these activities accordingly.
  - Can you switch to IEEE floating point data types from VAX floating-point data types? In general, applications can still use VAX floating point-format on I64. Use VAX floating point

formats when access to previously generated binary floating-point data is required. VAX floating point is achieved on I64 systems in software. VSI compilers 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 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. (For many compilers, specify the compiler option /FLOAT=IEEE. If using OpenVMS I64 BASIC, use the /REAL_SIZE qualifer.) If that produces acceptable results, you should simply build the application on the I64 system (and on 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.

- Backup and restore functionality
- Operator interface look and feel
- System administration

3.3. Additional Considerations

VSI recognizes that not all layered software and middleware supplier products will port to I64 at first release. VSI 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.

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

Chapter 4. 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.

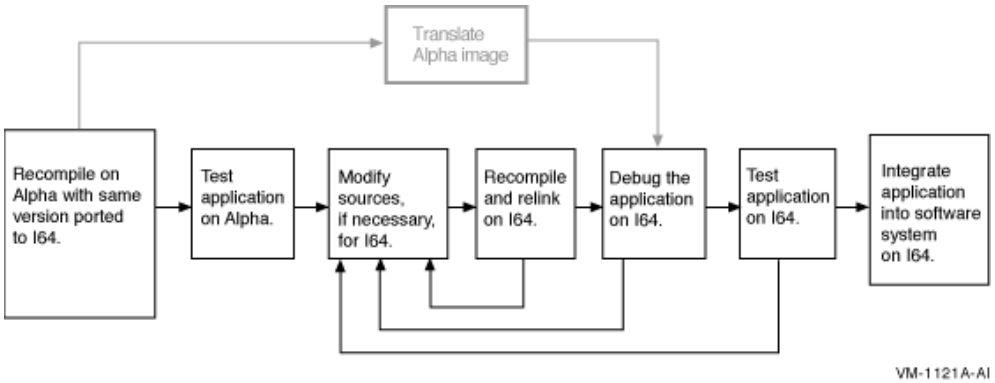
If you have thoroughly analyzed your code and planned the migration process, the final stage of migrating source modules should be fairly straightforward. You may be able to recompile or translate many programs with no change. Programs that do not recompile or translate directly will frequently need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

Figure 4.1. Migrating an Application from Alpha to OpenVMS I64



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 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 floating point-data types
- Use threads
- Contain unaligned data
- Rely explicitly on the OpenVMS Alpha calling standard
- 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 OpenVMS 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 OpenVMS I64 migration process is the availability of support from VSI Services, which can provide help in modifying, debugging, and testing your application..

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

- VSI rx2600 Integrity Server with one or two processors
- 1 to 24 GB of memory
- One to four 36-GB disks
- DVD/CD-ROM drive

For more information, refer to the *HP OpenVMS Version 8.2 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
    - Translation tools
        - VSI OpenVMS Migration Software for Alpha to Integrity Servers
        - TIE
    - RTLs
    - System libraries
    - Include files for C programs
- Logical names

  Logical names must be consistently defined to point to Alpha and OpenVMS 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 need to port the Alpha test tools to I64, unless they are already ported. You also need test tools that are designed to test for I64 characteristics, such as changes to the OpenVMS Calling Standard that are specific to I64.
- Tools for maintaining sources and building images, such as:
    - CMS (Code Management System)
    - MMS (Module Management System)

Native Alpha Development

The standard development tools you have on Alpha are also available as native tools on OpenVMS I64 systems.

Translation

The software translator VSI OpenVMS Migration Software for Alpha to Integrity Servers runs on both Alpha and OpenVMS I64 systems. The Translated Image Environment (TIE), which is required to run a translated image, is part of OpenVMS I64, so final testing of a translated image must either be done on an OpenVMS I64 system or at an OpenVMS I64 Migration Center. In general, the process of translating an Alpha image to run on an OpenVMS I64 system is straightforward, although you may have to modify your code somewhat to get it to translate without error.

4.1.2.1. VSI OpenVMS Migration Software for Alpha to Integrity Servers and Translated Image Environment (TIE)

The main tools for migrating Alpha user-mode images to I64 are a static translator and a run-time support environment:

- The VSI OpenVMS Migration Software for Alpha to Integrity Servers is a utility that analyzes an Alpha image and creates a functionally equivalent translated image. Using VSI OpenVMS Migration Software for Alpha to Integrity Servers, you will be able to do the following:
  - Determine whether an Alpha image is translatable.
  - Translate the Alpha image to an OpenVMS I64 image.
  - Identify specific incompatibilities with I64 within the image and, when appropriate, obtain information on how to correct those incompatibilities in the source files.
  - Identify ways to improve the run-time performance of the translated image.
- The Translated Image Environment (TIE) is an OpenVMS I64 shareable image that supports translated images at run time. TIE provides the translated image with an environment similar to OpenVMS Alpha and processes all interactions with the native OpenVMS I64 system. Items that TIE provides include:
  - Alpha instruction interpreter, which supports:
    - Execution of Alpha instructions (including instruction atomicity) that is similar to their execution on an Alpha system
    - Complex Alpha instructions, as subroutines
  - Alpha compatible exception handler
  - Jacket routines that allow communication between native and translated code
  - Emulated Alpha stack

  TIE is invoked automatically for any translated image; you do not need to call it explicitly.

VSI OpenVMS Migration Software for Alpha to Integrity Servers locates and translates as much Alpha code as possible into OpenVMS I64 code. TIE interprets any Alpha code that cannot be converted into OpenVMS I64 instructions; for example:

- Instructions that VSI OpenVMS Migration Software for Alpha to Integrity Servers could not statically identify
- VAX (F_, G_ and D_foating) floating-point operations

Since interpreting instructions is a slow process, VSI OpenVMS Migration Software for Alpha to Integrity Servers attempts to find and translate as much Alpha code as possible to minimize the need for interpreting it at run time. A translated image runs at slower than a comparable native Alpha image, depending on how much Alpha code TIE needs to interpret.

Note that you cannot specify dynamic interpretation of a Alpha image on an OpenVMS I64 system. You must use VSI OpenVMS Migration Software for Alpha to Integrity Servers to translate the image before it can run on I64.

Translating a Alpha image produces an image that runs as a native image on OpenVMS I64 hardware. The OpenVMS I64 image is not merely an interpreted or emulated version of the Alpha image, but contains OpenVMS I64 instructions that perform operations identical to those performed by the instructions in the original Alpha image. The I64.EXE file also contains the original Alpha image in its entirety, which allows TIE to interpret any code that VSI OpenVMS Migration Software for Alpha to Integrity Servers could not translate.

The VSI OpenVMS Migration Software for Alpha to Integrity Servers's analysis capability also makes it useful for evaluating programs that you intend to recompile, rather than translate.

4.1.3. Coexistence with Translated Images

Application components can be migrated from OpenVMS VAX and OpenVMS Alpha to I64 by binary translation. Because executable and shareable images can be translated on an individual basis, it is possible (even likely) to construct an environment in which native and translated images are mixed in a single image activation.

Translated images use the calling standard of their original implementation; that is, the argument lists of VAX translated images have the same binary format as they did on the VAX platform. Because the I64 calling standard differs from the OpenVMS VAX and OpenVMS Alpha calling standards, direct calls between native and translated images are not possible. Instead, such calls must be handled by interface routines (called jacket routines) that transform the arguments from one calling standard to the other.

The jacket routines are provided by TIE. In addition to argument transformation, TIE also handles the delivery of exceptions to translated code. TIE is loaded as an additional shareable image whenever a translated image (either main program or shareable image) is activated. The image activator interposes calls to jacket routines as necessary when it resolves references between native and translated images.

For native code to interoperate with translated code, it must meet the following requirements:

- All function descriptors must contain signature data. The signature data is used by the jacket routines to transform arguments and return values between their native and VAX or Alpha formats. Native OpenVMS I64 images contain function descriptors for both outbound calls and all entry points.
- All calls to function variables (that is, where the identity of the called function is contained in a variable) must be made through the library routine OTS$CALL_PROC. OTS$CALL_PROC determines at run time whether the function being called is native or translated and calls native code directly and translated code via the TIE.

These requirements are met by compiling code using the /TIE qualifier and linking it using the /NONATIVE_ONLY qualifier. /TIE is supported by most I64 compilers including the Macro-32 compiler. (It is not supported on C++.) Both qualifiers must be given explicitly in the compile and LINK commands, since the defaults are /NOTIE and /NATIVE_ONLY, respectively. Signatures are not supported by the OpenVMS I64 assembler; therefore, routines coded in OpenVMS I64 assembly language cannot be directly called from translated images. OpenVMS I64 assembly language can call translated code only with an explicit call to OTS$CALL_PROC.

Translated image support carries a minor performance penalty: all calls to function variables are made via OTS$CALL_PROC, at a cost of about 10 instructions for most cases. Performance-critical code should only be built /TIE and /NONATIVE_ONLY if interoperation with translated images is required.

An additional consideration applies to executable and shareable images that are linked /NOSYSSHR. Ordinarily, references to OTS$CALL_PROC are resolved by the shareable image LIBOTS.EXE and require no special attention. Linking an image /NOSYSSHR skips the search of the shareable image library and instead resolves external references from object modules in STARLET.OLB. OTS$CALL_PROC makes a reference to a data cell defined in the system symbol table, and if simply included from STARLET.OLB would result in an unresolved reference to CTL$GQ_TIE_SYMVECT.

Most images linked /NOSYSSHR are so built to avoid interaction with any external images, so calls to translated code are not an issue. For this reason, STARLET.OLB contains a subset version of OTS$CALL_PROC that does not support calls to translated images. This module, named

OTS$CALL_PROC_NATIVE, is loaded by default, and images linked /NOSYSSHR by default cannot call out to translated code.

In addition, STARLET.OLB also contains the module OTS$CALL_PROC, which is the full version. It has no entry in the library symbol table and is only loaded by explicit reference. For the rare cases where images linked /NOSYSSHR need to be able to call out to translated shareable images, the full version of OTS$CALL_PROC must be explicitly included from STARLET.OLB and the image must also be linked against the system symbol table. Two options are required in the link command file:

- SYS$LIBRARY:STARLET.OLB/INCLUDE=OTS$CALL_PROC must be present in the input file list.
- The /SYSEXE qualifier must be included on the LINK command.

4.2. Compiling Applications on Alpha With Current Compiler Version

Before recompiling your code with a native OpenVMS I64 compiler, VSI 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 OpenVMS 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 OpenVMS I64.

For a list of the native 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 OpenVMS I64. You can then compare the results of these tests with the results of similar tests on an OpenVMS I64 system, as described in Section 4.6.

4.4. Recompiling and Relinking on an OpenVMS I64 System

In general, migrating your application involves repeated cycles of revising, compiling, linking, and debugging your code. During the process, you 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 more information about the compiler switches, see Chapter 5. If you are porting VAX MACRO code, refer to the *VSI OpenVMS MACRO Compiler Porting and User's Guide*.

4.5. Debugging the Migrated Application

Once you have migrated your application to 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 *VSI OpenVMS Debugger Manual*.

4.5.1. Debugging

The I64 operating system provides the following debuggers:

- 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.

  The OpenVMS Debugger does not support debugging of translated images. However, you can debug a native application that uses a translated image.

- 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.

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 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 I64 systems.

To use SDA on an Alpha system, you must first familiarize yourself with the OpenVMS calling standard for Alpha systems. Similarly, to use SDA on an OpenVMS I64 system, you must familiarize yourself with the OpenVMS Calling Standard for OpenVMS I64 systems before you can decipher the pattern of a crash on the stack. For more information, refer to the *VSI 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 failure and are available only for the most recent failure, the summary crash history file with a separate listing file for each failure is a permanent record of system failures.

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 OpenVMS I64 system, there are two types of tests you should apply:

- 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 OpenVMS I64

Because the changes in your application are combined with use of a new architecture, testing your application after it is migrated to I64 is particularly important. Not only can the changes introduce errors into the application, but the new environment can 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 Alpha test suite along with the application (if the tests are not already available on OpenVMS I64).
3. Validate the test suite on an OpenVMS I64 system.
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 OpenVMS I64 Tests

Although 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, you may encounter bugs that were in your program all along but that never caused a problem on an OpenVMS Alpha system.

For example, failure to initialize some variable in your program might have been benign on an Alpha system but could produce an arithmetic exception on an OpenVMS I64 system. The same could be true of 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 and 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 OpenVMS 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 architecture environment, be sure that your application refers to the correct version.

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

    If you attempt to run an Alpha image on an OpenVMS 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 VSI 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 OpenVMS 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 a command definition file
- Code that uses threads, especially custom-written tasking or stack switching

- Privileged code

4.8.1. Conditionalized Code

This section describes how to conditionalize OpenVMS code when migrating to OpenVMS I64. This code will be compiled for both Alpha and OpenVMS I64, or for VAX, Alpha, and OpenVMS I64. The symbol ALPHA, referred to in the following sections, is new. However, the symbol EVAX has not been eliminated. You do not need 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 OpenVMS 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 OpenVMS I64 are undefined. On OpenVMS I64, OpenVMS 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 OpenVMS I64 systems.

For Alpha-specific code:

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

For OpenVMS I64-specific code:

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

4.8.1.2. BLISS Sources

For BLISS source files, either BLISS-32 or BLISS-64, the macros VAX, EVAX, ALPHA and OpenVMS I64 are defined in ARCH_DEFS.REQ. On Alpha, EVAX and ALPHA equal 1 while VAX and OpenVMS I64 equal 0. On OpenVMS I64, OpenVMS 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 OpenVMS I64 systems.

For Alpha-specific code:

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

For OpenVMS I64-specific code:

```
%if OpenVMS 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 OpenVMS I64-specific code, use the following:

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

4.8.1.4. Existing Conditionalized Code

Existing conditionalized code must be examined to determine whether 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 OpenVMS I64 code, then you would add the following case:

```
%IF OpenVMS I64 %THEN
```

```
  iii
  iii
%FI
```

However, if the existing VAX/EVAX conditionals really reflect 32 bits and not 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 OpenVMS I64 code are the same and 64-bit code 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 OpenVMS I64. The following sections describe system services and their replacement services.

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, different library routines return invocation context handles for I64. For more information, refer to the *VSI OpenVMS Calling Standard*.

SYS$GOTO_UNWIND is most frequently used to support programming language features, so changes are mostly in compilers or run-time libraries. However, any direct use of SYS$GOTO_UNWIND requires 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 4.8.9.

4.8.3. Code With Other Dependencies on the Alpha Architecture

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

4.8.3.1. Initialized Overlaid Program Sections

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

4.8.3.2. Condition Handlers Use of SS$_HPARITH

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

4.8.3.3. Mechanism Array Data Structure

The mechanism array data structure on I64 is very different from the one on OpenVMS Alpha. The return status code RETVAL has been extended to represent the return status register on both Alpha and OpenVMS I64 platforms. For more information, refer to the *VSI 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 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 OpenVMS 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 and image file features that are specific to the OpenVMS operating system, will be published in a future release.

The portion of an image which is used by the debugger 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 I64 is the industry-standard DWARF debug symbol table format described at this location. VSI 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. I64 supports IEEE floating-point in hardware and VAX floating-point data types in software.

Most of the 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.

to specify a default floating-point data types using the OpenVMS I64 BASIC compiler, you use the /REAL_SIZE qualifier. The possible values that can be specified are SINGLE (Ffloat), DOUBLE (Dfloat), GFLOAT, SFLOAT, TFLOAT, and XFLOAT.

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 OpenVMS I64 system using the same qualifier.

When you compile an OpenVMS application that specifies an option to use VAX floating-point on OpenVMS 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, as appropriate for the length.
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 certain applications.

For more information about the differences between floating-point data types on OpenVMS Alpha and I64 and how these differences might affect ported applications, see Chapter 5 and refer to the *OpenVMS Floating-Point Arithmetic on the Intel® Itanium ® Architecture* white paper.

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 might 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 nonzero 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. This default is the same default established by OpenVMS I64 at process startup.

4.8.4.1. LIB$WAIT Problem and Solution

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

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

On 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 OpenVMS 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 OpenVMS 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. Incorrect Command Table Declaration

An incorrect declaration of a command table in code ported to I64 can cause unexpected results. For example, for an application, the Command Definition Utility is used to create an object module from a CLD file. The application then calls CLI$DCL_PARSE to parse command lines. CLI$DCL_PARSE may fail with:

```
%CLI-E-INVTAB, command tables have invalid format - see documentation
```

The code must be modified so that the command table is defined as an external data object.

For example, if in an application, on VAX and Alpha, the command table (DFSCP_CLD) is incorrectly declared in a BLISS module as:

```
EXTERNAL ROUTINE DFSCP_CLD
```

This should be changed to

```
EXTERNAL DFSCP_CLD
```

Or if it was in a FORTRAN module incorrectly declared as:

```
EXTERNAL DFSCP_CLD
```

then it should be changed to

```
INTEGER DFSCP_CLD
CDEC$ ATTRIBUTES EXTERN :: DFSCP_CLD
```

Similarly, in an application written in C, if the command tables previously were defined as follows:

```
int jams_master_cmd();
```

The code should be changed to be an external reference:

```
extern void* jams_master_cmd;
```

The changed, correct declaration works on all platforms: VAX, Alpha or OpenVMS I64.

4.8.6. Code that Uses Threads

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 I64 without change. This section describes the exceptions. The major porting issue for code that uses threads is the usage of stack space. OpenVMS I64 code uses much more stack space than does equivalent Alpha code. Therefore, a threaded program that works on Alpha might get stack overflow failures on OpenVMS I64.

The default stack size is larger on 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 might need to be changed to request more stack. VSI recommends starting with an increase of three 8-Kb pages (24576 bytes).

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 might cause the process to exceed its memory quotas. In an extreme case, the P0 region could fill completely, in which case the process might need to reduce the number of threads in use concurrently (or make other changes to lessen the demand for P0 memory).

VSI recommends that you familiarize yourself with thread support in OpenVMS. 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.

The DCL command THREADCP is not supported on I64. For OpenVMS I64, the DCL commands SET IMAGE and SHOW IMAGE can be used to check and modify the state of threads-related image header flags, similar to the THREADCP command on OpenVMS Alpha. For details, refer to the *VSI OpenVMS DCL Dictionary*.

The THREADCP command is documented in the *Guide to POSIX Threads Library*.

If you want to change the setting of threads-related image flags, you need to use the new command SET IMAGE. For example:

```
$ SET IMAGE/FLAGS=(MKTHREADS,UPCALLS) FIRST.EXE
```

4.8.6.1. 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.

If a C language module (which uses either cma_delay or cma_time_get_expiration) is compiled by mistake in an IEEE floating-point mode, a compiler warning similar to the following will be displayed:

```
cma_delay (
    ^
%CC-W-LONGEXTERN, The external identifier name exceeds 31
characters; truncated to "CMA_DELAY_NEEDS_VAX_FLOAT______".
```

If an object file which triggered such a warning is linked, the linker will display an undefined-symbol message for this symbol. (If a linker-produced image was subsequently executed, the calls to these routines would fail with an ACCVIO.) These compiler and linker diagnostics are intended to alert you to the fact that these CMA routines require the use of VAX format floating-point values, and that the compilation was done in a manner that does not satisfy this requirement.

Note

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

4.8.7. Code With Unaligned Data

VSI 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 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, I64 systems provide help to manage the impact of unaligned data references. Alpha and OpenVMS 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 OpenVMS I64 compilers that allows the compiler to report compile-time references to unaligned data.

Table 4.1. Compiler Switches for Reporting Compile-Time Reference

Compiler	Switch
BLISS	/CHECK=ALIGNMENT
C	/WARN=ENABLE=ALIGNMENT
Fortran	/WARNING=ALIGNMENT
VSI Pascal	/USAGE=PERFORMANCE

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 must adapt all parts of the application to expect the same type of data, either aligned or unaligned.

4.8.8. Code that Relies on the OpenVMS Alpha Calling Standard

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

4.8.9. Privileged Code

This section describes categories of privileged code that require examination, and may require modifications.

4.8.9.1. Use of SYS$LKWSET and SYS$LKWSET_64

If your application uses SYS$LKWSET or SYS$LKWSET_64 to lock itself into memory, and your application does not run on VAX systems, consider replacing these calls with calls to the new (as of OpenVMS Version 8.2) LIB$LOCK_IMAGE RTL routine. Similarly, replace the SYS$ULWSET and SYS$ULWSET_64 calls with calls to the new LIB$UNLOCK_IMAGE RTL routine.

Programs that enter kernel mode and increase IPL to higher than 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 systems, 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 OpenVMS I64 systems, 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 SYS$LKWSET and SYS$LKWSET_64 system services on Alpha and OpenVMS I64.

As of OpenVMS Version 8.2, the SYS$LKWSET and SYS$LKWSET_64 system services 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 increase IPL to higher than 2 and without crashing 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 OpenVMS 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 (available in OpenVMS Version 8.2). 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.

4.8.9.2. Use of SYS$LCKPAG and SYS$LCKPAG_64

If your application uses SYS$LCKPAG or SYS$LCKAPG_64 to lock code in memory, examine your use of this service. On OpenVMS I64, this service will not lock the entire image in the working set.

It is likely that you intend to lock the image in the working set so your code can elevate IPL and execute without incurring a page fault. Refer to Section 4.8.9.1. See also the *VSI OpenVMS RTL Library (LIB$) Manual* for information about the LIB$LOCK_IMAGE and LIB$UNLOCK_IMAGE routines.

4.8.9.3. Terminal Drivers

The interface for terminal class drivers on 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 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`

4.8.9.4. Protected Image Sections

Protected image sections usually occur in shareable images which implement "User Written System Services."

These image sections are protected by software and hardware mechanisms to assure that an unprivileged application cannot compromise the integrity of these sections. Changes in hardware pages protection from VAX and Alpha to OpenVMS I64 have added some subtle restrictions which may require changes in the protected images.

As on VAX and Alpha, data sections that are writeable in privileged modes (kernel or exec) may be read by unprivileged (user) mode. The hardware protection for such pages does not allow execute access from any mode. Protected image sections which are linked as both writeable and executable are protected to allow inner mode read, write, and execute; no user mode access is allowed. As neither user mode access to inner mode writeable data, nor code being in writeable sections, is a common practice, it is felt that few applications are likely to require both in a single section.

While there were exceptions on VAX and Alpha, on OpenVMS I64, all writeable protected image sections on OpenVMS I64 are protected against user mode write. Protected images which intend to allow user write to protected image sections must use $SETPRT/$SETPRT_64 to alter the page protection.

Chapter 5. I64 Development Environment

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

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

5.1. Native OpenVMS I64 Compilers

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

- BASIC
- BLISS (available on the Freeware CD)
- C++
- COBOL V2.8
- Fortran V8.0 (Fortran 90)
- VSI C V6.5
- VSI Pascal
- Java
- VAX MACRO-32

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

The OpenVMS I64 compilers provide command-line qualifiers for using 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 I64

For new program development for I64, VSI recommends the use of high-level languages. VSI provides the VAX Macro-32 compiler for I64 to convert existing VAX MACRO code into machine code that runs on I64 systems.

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

- Programs that call routines that do not comply to the I64 calling standard.

- Programs that use the JSB instruction to call routines written in a language other than Macro-32.

For more information about porting VAX MACRO programs to I64 systems, refer to the *VSI 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 OpenVMS I64 applications. These tools are summarized in Table 5.1.

Table 5.1. OpenVMS Development tools

Tool	Description
OpenVMS Linker	The OpenVMS Linker accepts OpenVMS I64 object files to produce an OpenVMS I64 image. For more information about the OpenVMS Linker, see Section 5.3.
OpenVMS Debugger	The OpenVMS Debugger running on 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 OpenVMS 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 OpenVMS I64 images.
ANALYZE/OBJECT	The Analyze/Object utility can analyze OpenVMS 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	The Command Definition utility (CDU) enables application developers to create commands with a syntax similar to DCL commands.
System Dump Analyzer (SDA)	SDA has been extended to display information specific to I64 systems.
Crash Log Utility Extractor (CLUE)	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.

5.2.1. Translating Alpha Code

The VSI OpenVMS Migration Software for Alpha to Integrity Servers is a binary translator for translating Alpha user code images to run on I64 systems. This tool is useful in cases where the source

code is no longer available, or when the compiler is not available on OpenVMS I64, or when a third-party product has not been ported to I64. A third-party product's permission is required from the owner of the software.

The VSI OpenVMS Migration Software for Alpha to Integrity Servers offers functionality similar to that of DECmigrate, a binary translator that was used to port OpenVMS VAX images to run on OpenVMS Alpha systems.

For more information, see Section 4.1.2.1.

5.3. Linking Modules

The purpose of the linker is to create images (files that contain binary code and data). The linker ported to I64 is different from the linker on OpenVMS VAX and Alpha systems because it accepts I64 object files and produces 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 I64 Linker also creates **shareable images**. A shareable image is a collection of procedures and data that are exported via the symbol_vector option, that can be called by executable images or other shareable images. Shareable images are included in an input file via a link operation that creates an executable or shareable image.

When linking modules, the primary type of input file to the 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 Intel Itanium architecture, some aspects of linking modules on I64 are different. In general, the 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 I64 systems. These include:

- Differences when linking on OpenVMS I64 systems as compared with linking on VAX and Alpha systems (Section 5.3.1)
- Expanded linker map file information for OpenVMS I64 (Section 5.3.2)
- New linker qualifiers and options (Section 5.3.3)
- Mixed-Case Arguments in Linker Options, Revisited (Section 5.3.4)

For details of these features and considerations, refer to the *HP OpenVMS Version 8.2 New Features and Documentation Overview*.

5.3.1. Differences When Linking on I64 Systems

Although linking on I64 systems is similar to linking on OpenVMS Alpha systems, some differences exist. The following qualifiers or options are ignored by the I64 linker:

- /REPLACE
- /SECTION_BINDING
- /HEADER

- The per_page keyword in /DEMAND_ZERO=PER_PAGE (the per_page keyword will be supported, though with a slightly different meaning, in a future release)
- DZRO_MIN
- ISD_MAX

The following qualifiers and options are not allowed by the I64 Linker:

- /SYSTEM
- The file name keyword in /DEBUG=file_spec
- The base address keyword must be null in CLUSTER=cluster_name,base_address …
- BASE=
- UNIVERSAL=

The following are new qualifiers supported by the I64 Linker:

- /BASE_ADDRESS
- /SEGMENT_ATTRIBUTE
- /FP_MODE
- /EXPORT_SYMBOL_VECTOR
- /PUBLISH_GLOBAL_SYMBOLS
- The GROUP_SECTIONS and SECTIONS_DETAILS keywords for the /FULL command

Some of these qualifiers and options are described in the following sections. For more details on these qualifiers and options see the *HP OpenVMS Version 8.2 New Features and Documentation Overview*.

5.3.1.1. No Based Clusters

Specifying a base address in a CLUSTER option is permitted on VAX and Alpha. On VAX even shareable images are allowed to contain based clusters, whereas on Alpha only main images are allowed to consist of such clusters. For OpenVMS I64 specifying a base address in a CLUSTER option is illegal for all images.

5.3.1.2. Handling of Initialized Overlaid Program Sections on I64

On Alpha and VAX systems, initializations can be done to portions of an overlaid program section. Subsequent initializations to the same portions overwrite initializations from previous modules. The last initialization performed on any byte is used as the final one of that byte for the image being linked.

On OpenVMS I64 systems, the ELF (Executable and Linkable Format) object language does not implement the feature of the Alpha and VAX object language which allows the initialization of portions of sections. When an initialization is made, the entire section is initialized. Subsequent initializations of this section may be performed only if the non-zero portions match in value.

For example, the following condition produces different results on I64 systems than on Alpha systems:

Two program sections (simply termed 'sections' in ELF), each declaring two longwords, are overlaid. The first program section initializes the first longword and the second program section initializes the second longword with a non-zero value.

On Alpha systems, the linker is able to produce an image section with the first and second longword initialized. The VAX and Alpha object languages give the linker the section size and the Text Information Relocation (TIR) commands to initialize the section.

On OpenVMS I64 systems, the linker gets pre-initialized sections which contain initialization data from the compilers. The linker does not perform or know about the initializations since there are no TIR commands in ELF. The linker then produces a segment using the last processed section for initialization. That is, in the image either the first or second longword has a non-zero value, depending on the order in which the modules containing the sections were linked. On OpenVMS I64 systems, the linker reads the sections to be overlaid and checks for compatibility. Any two overlaid sections are compatible if they are identical in the non-zero values. If they are not compatible, the linker issues the following error:

```
%ILINK-E-INVOVRINI, incompatible multiple initializations for overlaid
 section
    section: <section name>
    module: <module name for first overlaid section>
    file: <file name for first overlaid section>
    module: <module name for second overlaid section>
    file: <file name for second overlaid section>
```

In the previous message, the linker lists the first module that contributes a non-zero initialization, and the first module with an incompatible initialization. Note that this is not a full list of all incompatible initializations; it is just the first one the linker encounters.

In the Program Section Synopsis of the linker map, each module with a non-zero initialization is flagged as "Initializing Contribution." Use this information to identify and resolve all the incompatible initializations. For further examples and more detail on the handling of initialized overlayed sections, see the *HP OpenVMS Version 8.2 New Features and Documentation Overview*.

5.3.1.3. Behavior Difference When Linking ELF Common Symbols

The OpenVMS I64 linker behaves differently when ELF common symbols (also known as relaxed ref/def symbols) are linked selectively against an image that contains a definition for the same symbol. On Alpha, the linker incorrectly takes the definition from the relaxed ref/def symbol in your module. The OpenVMS I64 linker takes the definition from the shareable image.

5.3.2. Expanded Map File Information

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

- On Alpha systems, the map section that was titled Object Module Synopsis has been renamed on OpenVMS 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 OpenVMS I64 systems: Cluster Synopsis and Image Segment Synopsis. The information has been moved to the appropriate section.
- In the Program Section Synopsis, the PIC and NOPIC attributes, invalid on OpenVMS 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 relocateable and external. However, the linker does not know whether an external symbol is relocatable or not. As a result, on OpenVMS 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, (very likely only used by C++), designated by the UxWk key, are a new addition to I64.

For an example linker map illustrating this new information, refer to the *HP OpenVMS Version 8.2 New Features and Documentation Overview*.

5.3.3. New Linker Qualifiers and Options for I64

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

5.3.3.1. New /BASE_ADDRESS Qualifier

A new qualifier, /BASE_ADDRESS, is provided on OpenVMS I64 systems. The base address is the starting address that you want the linker to assign to an executable image. The purpose of this qualifier is to assign a virtual address for images which are not activated by the OpenVMS image activator, such as images used in the boot process. The OpenVMS image activator is free to ignore any linker assigned starting address. This qualifier is primarily used by system developers.

The /BASE_ADDRESS qualifier does not replace the CLUSTER=,*[base-address]* option, which is illegal on I64. See Section 5.3.1.1.

For more information on this qualifier, see the *HP OpenVMS Version 8.2 New Features and Documentation Overview*.

5.3.3.2. New /SEGMENT_ATTRIBUTE Qualifier

The I64 Linker provides a new /SEGMENT_ATTRIBUTE which accepts two keywords: SHORT_DATA=WRITE and DYNAMIC_SEGMENT = P0 or P1. The DYNAMIC_SEGMENT keyword is rarely needed. For more information see the *HP OpenVMS Version 8.2 Release Notes*.

The SHORT_DATA=WRITE keyword allows you to combine read-only and read-write short data sections into a single segment, reclaiming up to 65,535 bytes of unused read-only space. When setting SHORT_DATA to WRITE, your program may accidentally write to formerly read-only data. Therefore this qualifier is only recommended for users whose short data segment has reached the limit of 4 MB.

For more details on this qualifier, see the *HP OpenVMS Version 8.2 Release Notes*.

5.3.3.3. New /FP_MODE Qualifier

The I64 Linker determines the program's initial floating point mode using the floating point mode provided by the module that provides the main transfer address. Use the /FP_MODE qualifier to set an initial floating point mode only if the module that provides the main transfer address does not provide an initial floating point mode. The /FP_MODE qualifier will not override an initial floating point mode provided by the main transfer module. The OpenVMS I64 Linker accepts the following keywords to set the floating point mode:

- D_FLOAT, G_FLOAT – Sets VAX floating point modes.

- IEEE_FLOAT[=ieee_behavior] – Sets the IEEE floating point mode to the default or a specific behavior.

The I64 Linker accepts the following IEEE behavior keywords:

- FAST
- UNDERFLOW_TO_ZERO
- DENORM_RESULTS (default)
- INEXACT

The I64 Linker also accepts a floating point mode behavior literal. For more information about the initial floating point mode, see the *VSI OpenVMS Calling Standard*.

5.3.3.4. New /EXPORT_SYMBOL_VECTOR and /PUBLISH_GLOBAL_SYMBOLS Qualifiers

The EXPORT_SYMBOL_VECTOR and PUBLISH_GLOBAL_SYMBOLS qualifiers were added to the linker to aid customers who are creating shareable images but did not know which symbols to export through the SYMBOL_VECTOR option. This could be because they were porting an application from UNIX and were not familiar enough with the application to know which symbols to export, or because they were coding in C++, and were not able to know what the mangled names looked like.

The I64 Linker provides a new /PUBLISH_GLOBAL_SYMBOLS qualifier to mark an object module, so that all its global symbols can be exported in a symbol vector option. Additionally, the I64 Linker provides a new /EXPORT_SYMBOL_VECTOR qualifier to export a symbol vector option and to name the output file.

Both qualifiers are only accepted if the /SHAREABLE qualifier is present. /EXPORT_SYMBOL_VECTOR is a command line only qualifier. /PUBLISH_GLOBAL_SYMBOLS can be used in option files as well. The linker will warn, if there is an /EXPORT_SYMBOL_VECTOR qualifier but no /PUBLISH_GLOBAL_SYMBOLS qualifier was seen.

When /EXPORT_SYMBOL_VECTOR is present, only the option file is written, no image file is generated. The generated option file needs to be completed with GSMATCH information by the developer.

The /PUBLISH_GLOBAL_SYMBOLS qualifier is a positional qualifier and can be used with object files and libraries. It is compatible with the /INCLUDE and /SELECTIVE qualifiers, for example:

```
$ link/SHARE public/PUBLISH,implementation/EXPORT=public
$ link/SHARE plib/LIBRARY/PUBLISH/INCLUDE=public/EXPORT=public
```

The /EXPORT_SYMBOL_VECTOR qualifier is a positional qualifier, which accepts an optional output file specification. If there is no file name specified the linker will use the name of the last input file and construct a file name, default file type is ".OPT".

For more detailed information on this qualifier, see the *HP OpenVMS Version 8.2 New Features and Documentation Overview*.

5.3.3.5. 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 (that is, a 32-byte or 16-word alignment).

5.3.3.6. New GROUP_SECTIONS and SECTION_DETAILS keywords for the /FULL Qualifier

The I64 Linker takes two keywords to the /FULL qualifier. The first keyword, GROUP_SECTIONS, prints all of the groups that were used in the map. Today the only compiler that takes advantage of groups is C++. Using this keyword with other languages has no effect.

When /FULL=NOSECTION_DETAILS is specified the I64 Linker suppresses zero length contributions in the Program Section Synopsis of the map. When the qualifier /FULL is used, it defaults to /FULL=SECTION_DETAILS, and a full linker map on VAX, Alpha, and OpenVMS I64 systems lists all the module contributions in the Program Section Synopsis.

5.3.4. Mixed-Case Arguments in Linker Options, Revisited

On I64 systems, names issued by compilers may be mixed case names. If you need to operate on mixed-case names in the options file (for example you have a library include statement and the module names are mixed-case) the linker already has an option to process the names in mixed-case, rather than using its default behavior (uppercasing all names). That option is the CASE_SENSITIVE option as shown below:

```
CASE_SENSITIVE=YES.
```

When the CASE_SENSITIVE option is set to YES, all characters to the right of the left-most equal sign (such as option arguments) have their case preserved. In other words, these characters are taken "as-is" without modification. This includes file names, module names, symbol names and keywords. To restore the linker's default behavior of upcasing the entire option line, specify the CASE_SENSITIVE option with the NO keyword as follows:

```
CASE_SENSITIVE=NO
```

Note that the NO keyword must appear in uppercase or it will not be recognized by the linker.

To maintain VAX and Alpha behavior it is recommended to switch to case sensitivity only when needed.

For more information and examples, refer to the *HP OpenVMS Version 8.2 New Features and Documentation Overview*.

5.4. Debugging Capabilities on 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 (not available on I64)
- XDelta
- Source code debugger (SCD) (not available on I64)

The OpenVMS Debugger and XDelta debugger are available on OpenVMS 8.2. 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	no [1]
Supervisor mode	no	no [1]
Exec mode	no	yes
Kernel mode	no	yes

[1]IPLs 0, 1, and 2 only

The following sections describe the capabilities of the OpenVMS Debugger and the XDelta debugger running on I64 systems. Further information about resolved problems, limitations, and known problems of the OpenVMS Debugger can be found in the *HP OpenVMS Version 8.2 Release Notes*.

5.4.1. OpenVMS Debugger

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

5.4.1.1. Architecture Support

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

I64 Debugger supports programs written in the following languages:

- BLISS

- C
- C++ (limited)
- COBOL (limited)
- Macro-32
- Fortran
- Intel ® Assembler (IAS)
- Pascal

Some issues exist when debugging supported languages on I64. For information about these issues and suggested workarounds, refer to the *HP OpenVMS Version 8.2 Release Notes*.

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 Version 8.2 Release Notes*.

Macro-32 is a compiled language on 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. For complete information on register mapping used by Macro-32, refer to the *HP OpenVMS Version 8.2 Release Notes*.

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 command (kept debugger configuration)
- RUN/DEBUG command (normal debugger configuration)
- Multithreaded 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 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 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 I64 and OpenVMS Alpha systems.

5.4.2.1. XDelta Capabilities on I64

New capabilities have been added for 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
- ;T display interrupt stack command

For descriptions of these registers and capabilities in this release, refer to the *HP OpenVMS Version 8.2 New Features and Documentation Overview*. This document also describes the ;D and ;T commands, along with some restrictions.

5.4.2.2. Differences Between XDelta on I64 and OpenVMS Alpha Systems

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

Interrupting a Running System

To interrupt a running system on OpenVMS 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 lower than 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. I64 Librarian Utility

The Librarian utility on I64 provides the same features provided by Librarian on OpenVMS Alpha, with some changes and restrictions. This section describes what is unique about the OpenVMS I64 Librarian. For information about restrictions or other temporary conditions, refer to the *HP OpenVMS Version 8.2 Release Notes*.

5.5.1. Considerations When Using the OpenVMS I64 Librarian

You can use the DCL command LIBRARY (or Librarian LBR routines) to create libraries such as an OpenVMS I64 (ELF) object library, OpenVMS 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 OpenVMS I64 Librarian cannot create or process Alpha and VAX objects and shareable images. Rather, the architecture for the Librarian is Intel Itanium.

There is no architecture switch for OpenVMS I64 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 OpenVMS I64 systems:

LBR$C_TYP_ELFOBJ (9) — Represents an ELF object library.
LBR$C_TYP_ELFSHSTB (10) — Represents an ELF shareable image library.

In addition, the following library types for the LBR$OPEN routine are not supported in the OpenVMS 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_ESHSTB (8) - Represents an Alpha shareable image library

5.5.3. OpenVMS I64 Library Format Handles UNIX-Style Weak Symbols

Due to the requirements of the Intel C++ compiler, the OpenVMS 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 OpenVMS-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 6.0) applies only 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 encounter any change in behavior.

5.5.3.2. Version 6.0 Library Index Format

VSI recommends using the new Version 6.0 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. For more information, refer to the *HP OpenVMS Version 8.2 New Features and Documentation Overview*.

5.5.3.3. 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.

5.5.3.4. Current Library Limitation with Regard to Weak and Group Symbols

Library symbol entries are associated with the module within which they were defined. On I64, more than one module can define a UNIX-style weak or Group symbol. As such, the OpenVMS 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 (you can find more information about precedence ordering rules in the *HP OpenVMS Version 8.2 Release Notes*.)

Because of current limitations described in the *HP OpenVMS Version 8.2 Release Notes*, VSI suggests that you only perform insert operations into the library only for modules that contain UNIX-style weak definitions. If you 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.

Chapter 6. Preparing to Port Applications

This chapter provides an overview of porting considerations related to the primary compilers that are available for I64. Table 6.1 maps the versions of OpenVMS Alpha compilers that were used to port to the I64 versions.

Table 6.1. Mapping OpenVMS Alpha Compiler Versions to OpenVMS I64 Compiler Versions

Compiler	OpenVMS Alpha	OpenVMS I64	For more information
BASIC	V1.6	V1.6	See Section 6.2.
BLISS	V1.11-004	V1.12-067	See Section 6.3.
COBOL	V2.8	V2.8	See Section 6.4.
Fortran 77	—	Not available[1]	See Section 6.5.2.
Fortran 90	V7.5	V8.0	See Section 6.5.
GNAT Pro Ada 95	Third party	—	See Section 6.1.
VSI Ada 83	V3.5A	Not available [1]	—
VSI C	V6.5	V7.1	See Section 6.6.
VSI C++	V6.5	7.1	Section 6.7
VSI Pascal	V5.9	V5.9	Section 6.10
Java	1.4.2	1.4.2-1	See Section 6.8.
Macro-32	V4.1-18	T1.0-77	See Section 6.9.
Macro-64	V1.2	Not available [1]	—

[1]Compiler is platform specific (for OpenVMS Alpha only).

Most of the 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-point data types instead of VAX floating-point data types.

  The *OpenVMS Floating-Point Arithmetic on the Intel ® Itanium ® Architecture* white paper describes how OpenVMS I64 deals with floating-point issues.

VSI recommends that customers first compile their applications on Alpha with the versions of the compilers that are being ported to 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, linked, and run without error on the new compiler on an Alpha system, it can be ported to I64.

6.1. Ada

Ada 95 is available on OpenVMS I64 in OpenVMS Version 8.2. Ada 83 is not supported on OpenVMS I64.

6.2. BASIC

The same BASIC is supported on both OpenVMS Alpha and OpenVMS I64. See the *VSI BASIC Release Notes* for additional information on using BASIC on I64.

The BASIC compiler on Alpha defaults to /REAL_SIZE=SINGLE (VAX F-float), and on I64 it defaults to /REAL_SIZE=SFLOAT.

The BASIC compiler does not support the /IEEE_MODE qualifier. The compiler and RTL set up the run-time environment on OpenVMS I64 to be similar in terms of exception handling and rounding to what is provided on OpenVMS Alpha.

6.3. BLISS Compiler

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

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

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

The BLISS-32*xx* 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-64*xx* 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**
OpenVMS Alpha	BLISS-32EN	BLISS/A32 or BLISS
OpenVMS Alpha	BLISS-64EN	BLISS/A64
OpenVMS I64	BLISS-32IN	BLISS/I32 or BLISS
OpenVMS I64	BLISS-64IN	BLISS/OpenVMS I64

6.3.1. BLISS File Types and File Location Defaults

This section discusses file types and output file location defaults for the BLISS compiler.

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

Output File Location Defaults

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

In both OpenVMS Alpha and OpenVMS 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 [FOO]BAR/OBJ     ! Puts BAR.OBJ in directory FOO
$ BLISS /I32 [FOO]BAR/OBJ     ! Puts BAR.OBJ in directory FOO
$
$ BLISS /A32 /OBJ [FOO]BAR    ! Puts BAR.OBJ in default directory
$ BLISS /I32 /OBJ [FOO]BAR    ! Puts BAR.OBJ in default directory
$
$ BLISS /A32 [FOO]BAR/OBJ=[]  ! Puts BAR.OBJ in default directory
$ BLISS /I32 [FOO]BAR/OBJ=[]  ! Puts BAR.OBJ in default directory
```

6.3.2. OpenVMS Alpha BLISS Features Not Available

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

OpenVMS Alpha BLISS Machine-Specific Built-ins

Support for the following OpenVMS Alpha BLISS machine-specific built-ins is no longer available:

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 the section called “Compare and Swap Built-in Functions”.

OpenVMS Alpha BLISS PALcode Built-in Functions

Support for the following OpenVMS Alpha BLISS PALcode built-ins is no longer available:

```
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_DRAINA        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 are placed in STARLET.REQ for PALCALL built-ins. OpenVMS supplies the supporting code. The privileged CALL_PALs call executive routines and the unprivileged CALL_PALs will call system services.

OpenVMS Alpha BLISS Register Names

The following registers (whose default OpenVMS I64 use is indicated in the following list) 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
```

INTERRUPT and EXCEPTION Linkages

INTERRUPT and EXCEPTION linkages are not supported.

BUILTIN R*n*

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

6.3.3. I64 BLISS Features

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

I64 BLISS Built-ins

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

Common BLISS Built-ins

The following existing common BLISS built-ins 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
```

RETURNADDRESS Built-in

A new built-in function RETURNADDRESS returns the PC of the caller's caller.

This built-in takes no arguments. The format is:

```
RETURNADDRESS()
```

Machine-Specific Built-ins

The following OpenVMS Alpha BLISS machine-specific built-ins are supported on the OpenVMS I64 BLISS compiler:

```
BARRIER
ESTABLISH
REVERT

ROT
SLL
SRA
SRL
UMULH

AdaWI

ADD_ATOMIC_LONG   AND_ATOMIC_LONG  OR_ATOMIC_LONG
ADD_ATOMIC_QUAD   AND_ATOMIC_QUAD  OR_ATOMIC_QUAD
```

The *xxx*_ATOMIC_*xxx* built-ins no longer support the optional retry-count input argument. See the section called “ADD, AND, Built-in Functions for Atomic Operations” for details.

```
TESTBITSSI TESTBITCC  TESTBITCS
TESTBITCCI TESTBITSS  TESTBITSC
```

TESTBIT*xx* instructions no longer support the optional retry-count input argument or the optional success-flag output argument. See the section called “TESTBITxxI and TESTBITxx Built-in Functions for Atomic Operations” for details.

```
ADDD    DIVD    MULD    SUBD    CMPD
ADDF    DIVF    MULF    SUBF    CMPF
ADDG    DIVG    MULG    SUBG    CMPG
ADDS    DIVS    MULS    SUBS    CMPS
ADDT    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
```

New Machine-Specific Built-ins

A number of new built-ins added to I64 BLISS provide access to single OpenVMS I64 instructions that can be used by the operating system.

Built-ins for Single OpenVMS I64 Instructions

Each name that is capitalized in the following list is a new built-in function that can be specified. The lowercase name in parenthesis is the actual OpenVMS I64 instruction executed. The arguments to these instructions (and therefore their associated BLISS built-in 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 built-in 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, can be any expression whose value is moved into IPF general register R17 just prior to executing the BREAK instruction.

Access to Processor Registers

The I64 BLISS compiler provides built-in functions for access to read and write the many and varied processor registers in the IPF implementations. The built-in functions are as follows:

- GETREG
- SETREG
- GETREGIND
- SETREGIND

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

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

PALcode Built-ins

Support for the following OpenVMS Alpha BLISS PALcode built-in 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 SYS$PAL_*xxxx* run-time routine.

BLI$CALLG

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

OpenVMS I64 Registers

The OpenVMS 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 OpenVMS I64 general registers R40 through R127.

Naming of any of the OpenVMS 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.

ALPHA_REGISTER_MAPPING Switch

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

This switch can be specified in either the MODULE declaration or a SWITCHES declaration. Use of this switch causes a remapping of OpenVMS Alpha register numbers to OpenVMS 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 I64 BLISS (see the section called “OpenVMS Alpha BLISS Register Names” and the section called “OpenVMS I64 Registers”). 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, OpenVMS 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 OpenVMS 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
```

ALPHA_REGISTER_MAPPING and "NOTUSED"

When ALPHA_REGISTER_MAPPING is specified, any OpenVMS Alpha register that maps to an IA64 scratch register and is specified as NOTUSED in a linkage declaration will be placed in the PRESERVE set.

This will cause the register to be saved on entry to the routine declaring it NOTUSED and restored on exit.

/ANNOTATIONS Qualifier

The 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 negotiable. The qualifier itself is also negotiable. By default it is not present in the command line.

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

/ALPHA_REGISTER_MAPPING Qualifier

The 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.

/ALPHA_REGISTER_MAPPING Informationals

For 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, OpenVMS 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:

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

ADD, AND, Built-in Functions for Atomic Operations

The ADD_ATOMIC_*XXXX*, AND_ATOMIC_*XXXX*, and OR_ATOMIC_*XXXX* built-in functions for atomic updating of memory are supported by I64 BLISS. They are listed in the section called “Machine-Specific Built-ins”.

Because the OpenVMS I64 instructions to support these built-ins waits until the operation succeeds, the optional retry-count input parameter has been eliminated. These built-ins 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 Alpha BLISS optional retry-count parameter results in a syntax error.

TESTBITxxI and TESTBITxx Built-in Functions for Atomic Operations

The TESTBIT*xx*I and TESTBIT*xx* built-in functions for atomic operations are supported by I64 BLISS. They are listed in the section called “Machine-Specific Built-ins”.

Because the OpenVMS I64 instruction to support these built-ins waits until the operation succeeds, the optional input parameter retry-count and the optional output parameter success_flag have been eliminated. These built-ins now have the following form:

```
TESTBITxxx( field )
```

Any attempt to use the Alpha BLISS optional retry-count or success_flag parameters results in a syntax error.

Granularity of Longword and Quadword Writes

I64 BLISS supports the /GRANULARITY=*keyword* qualifier, the switch DEFAULT_GRANULARITY=*n*, and the data attribute GRANUALRITY(*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 either BYTE, LONGWORD, or QUADWORD. The parameter n must be either 0(byte), 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.

Shift Built-in Functions

Built-in functions for shifts in a known direction are supported for I64 BLISS. They are listed in the section called “Machine-Specific Built-ins”. These functions are valid only for shift amounts in the range 0..%BPVAL-1.

Compare and Swap Built-in Functions

Both Alpha and I64 BLISS provide support for the following new compare and swap built-in 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).

OpenVMS I64-Specific Multimedia Instructions

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

Linkages

CALL Linkage

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

JSB Linkage

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

/[NO]TIE Qualifier

Support for this qualifier continues for I64. The default is /NOTIE.

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.
- 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.

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

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

For I64 BLISS, the /ENVIRONMENT=NOFP command qualifier or ENVIRONMENT(NOFP) switch does not totally disable floating point because of the architectural features of OpenVMS I64. Instead, source code is still restricted not to 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 using a small subset of the floating-point registers. Specifically, if this option is specified, the compiler is restricted to using f6-f11, and sets the ELF EF_IA_64_REDUCEFP option described in the *Intel Itanium Processor-Specific Application Binary Interface*.

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

Floating-Point Support

The following sections discuss support for floating-point operations using the BLISS compiler.

Floating-Point Built-in 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 built-ins for floating-point arithmetic and conversion operations. For a complete list. see the section called “Machine-Specific Built-ins”.

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

Floating-Point Literals

The floating-point literals supported by I64 BLISS are the same set supported by Alpha BLISS: %E, %D, %G, %S and %T.

Floating-Point Registers

Direct use of the OpenVMS I64 floating-point registers is not supported.

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 I64 BLISS.

New and Expanded Lexicals

BLISS has added the following new compiler-state lexicals to support the 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 OpenVMS Alpha compilers.

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

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

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

- The lexicals %BLISS32I and %BLISS64I have been added. Their behavior parallels that of the new parameters to %BLISS.

- Support for the Intel Itanium architecture as a keyword to the %HOST and %TARGET lexicals has been added for I64 BLISS.

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.

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 it 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 then 8 bytes ignore the SHORT attribute.
- Data objects in the CODE, INITIAL and LINKAGE storage classes ignore the SHORT attribute, regardless of their size.
- 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

The following example shows the use of PSECT in BLISS code.

```
PSECT

  NODEFAULT = $GLOBAL_SHORT$
    (READ,WRITE,NOEXECUTE,NOSHARE,NOPIC,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,NOPIC,CONCATENATE,LOCAL,ALIGN(3),
    SHORT($GLOBAL_SHORT$)),

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

  PLIT = $PLIT$
    (READ,NOWRITE,NOEXECUTE,SHARE,NOPIC,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
```

Note

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

6.4. COBOL

COBOL Version 2.8 is supported on both OpenVMS Alpha and OpenVMS I64. See the *VSI COBOL Release Notes* for restrictions and known problems related to using COBOL on I64.

6.4.1. Floating-Point Arithmetic

The COBOL compiler on Alpha defaults to /FLOAT=D_FLOAT. For OpenVMS I64, the default is /FLOAT=IEEE_FLOAT.

The COBOL compiler does not support /IEEE_MODE. The COBOL RTL sets up the run-time environment on OpenVMS I64 to be similar in terms of exception handling and rounding to what is provided in the COBOL run-time environment on OpenVMS Alpha.

The COBOL Release Notes and the white paper entitled *OpenVMS Floating-Point Arithmetic on the Intel® Itanium ® Architecture* together describe how COBOL deals with floating-point issues on OpenVMS I64.

6.4.2. /ARCH and /OPTIMIZE=TUNE Qualifiers

For the sake of "compile-and-go" compatibility, OpenVMS Alpha values for the /ARCH and /OPTIMIZE=TUNE qualifiers are accepted by the COBOL compiler on OpenVMS I64. An informational message is displayed indicating that they are ignored.

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

6.5. Fortran

Fortran 90 V8.0 is supported on I64. Fortran 77 is not supported on I64. (See Section 6.5.2 for details.)

VSI Fortran for I64 systems provides the same command-line options and language features as VSI Fortran for Alpha systems with a few exceptions. These exceptions are discussed in the following sections.

6.5.1. Floating-Point Arithmetic

The Fortran release notes and the white paper entitled *OpenVMS Floating-Point Arithmetic on the Intel ® Itanium ® Architecture* together describe how VSI Fortran for OpenVMS I64 deals with floating-point issues.

See the Related Documents section in the Preface for the web location of this white paper and other OpenVMS-to-Itanium architecture resources.

Highlights are summarized as follows:

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

  The Alpha compiler defaults to /FLOAT=G_FLOAT. On 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 OpenVMS 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 OpenVMS Alpha.

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

Note that the changed /FLOAT default has implications for the use of /CONVERT=NATIVE (the default). This switch causes unformatted data to remain unconverted on input, on the assumption that it matches the prevailing floating-point datatype.

Files written from Fortran applications built with /FLOAT=G_FLOAT/CONVERT=NATIVE on OpenVMS Alpha (either explicitly G_FLOAT and NATIVE or defaulting either or both) will be in G_FLOAT. They can be read by OpenVMS I64 applications built with /FLOAT=G_FLOAT/CONVERT=NATIVE or /FLOAT=IEEE/CONVERT=VAXG. But they will not be readable by applications build with /CONVERT=NATIVE, as that will default to the /FLOAT type, which will have defaulted to IEEE.

- The /IEEE_MODE qualifier defaults to /IEEE_MODE=DENORM_RESULTS.

  This differs from OpenVMS 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 OpenVMS I64 (or on OpenVMS Alpha processors from EV6 onward).

- Users must choose one /FLOAT value and one /IEEE_MODE value and stick with it for the whole of their application. This is because mixed-mode applications do not (in general) work on I64 systems as a result of architectural differences in the hardware. This is a change from OpenVMS on OpenVMS Alpha systems, where mixed-mode applications work. In particular, per-routine/per-file/per-library settings of a mode do not work.

  As on OpenVMS Alpha, /IEEE_MODE can only be set if the user has chosen or defaulted to /FLOAT=IEEE. The IEEE_MODE used for G_FLOAT and D_FLOAT is one the compiler has picked as appropriate for supporting the IEEE-format calculations which implement VAX-format support.

- 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 might lead to unexpected results, since the compilers assume that no called routine changes the current mode. This is also a requirement on user-written libraries.

- Exception handlers are entered with the floating-point mode in effect at the time the exception was raised, not the mode with which the handler was compiled.

As specified in the white paper, *OpenVMS Floating-Point Arithmetic on the Itanium® Architecture*, the number, type and location of floating-point exceptions raised during the execution of an application on OpenVMS I64 may not be the same as on OpenVMS Alpha. This is particularly true for VAX-format floating-point. In that case exceptions are (in general) only raised on the convert back to VAX-format after the computation is over.

Three consequences may be of interest for users of VAX-format floating-point:

- Intermediate values (such as the "X/Y" in "X*Y + X/Y" which could have raised an exception on OpenVMS Alpha will probably not raise an exception on OpenVMS I64.

- Values not converted back to VAX-format floating-point (such as the "X/Y" in "IF (X/Y > 0.0)" will probably not raise an exception on OpenVMS I64.
- Alpha and VAX Fortran applications do not report underflows for VAX-format floating-point operations unless you specifically enable underflow traps by compiling with the /CHECK=UNDERFLOW qualifier. The same is true on OpenVMS I64 systems, but with an important caveat.

  Since all OpenVMS I64 floating-point operations are implemented using IEEE-format operations, enabling underflow traps with /CHECK=UNDERFLOW causes exceptions to be raised when values underflow the IEEE-format representation, not the VAX-format one.

  This can result in an increased number of underflow exceptions seen with /CHECK=UNDERFLOW when compared with equivalent OpenVMS Alpha or VAX programs, as the computed values may be in the valid VAX-format range, but in the denormalized IEEE-format range.

6.5.2. 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 OpenVMS Alpha F77 compiler that is not available in the OpenVMS I64 and OpenVMS Alpha F90 compilers:

- CDD and FDML support, which were previously available only in the OpenVMS Alpha F77 compiler, have now been implemented in the OpenVMS I64 Fortran 90 compiler.
- Code previously compiled with the F77 compiler that does not compile with the F90 compiler should be reported through the problem reporting mechanism.

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.

6.5.3. /ARCH and /OPTIMIZE=TUNE Qualifiers

For the sake of "compile-and-go" compatibility, OpenVMS 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.

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

6.6. VSI C/ANSI C

VSI C Version 7.1 is supported on OpenVMS I64 systems. Refer to the VSI C documentation, including the *VSI C Release Notes* for details and additional porting considerations.

6.6.1. OpenVMS I64 Floating-Point Default

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

On 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 OpenVMS 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 OpenVMS Alpha.

Note also that the default /IEEE_MODE has changed from FAST (on Alpha) to DENORM_RESULTS on I64. This means that, by default, floating-point operations 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 nonzero 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.

6.6.2. Semantics of /IEEE_MODE Qualifier

On 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 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 compiler marks the object module of each compilation with the floating-point control options specified by the compile-time qualifiers. When the OpenVMS I64 linker produces an executable image, it copies the floating point controls from the object module that supplied the main entry point transfer address for the image into the image itself; this is called the "whole program floating-point mode" for the image. Then when the image is activated for execution, the hardware's floating-point controls are initialized according to this whole program floating point mode. It is expected that code that modifies the floating-point controls at run-time will be written to ensure that the whole program floating point mode settings get restored whenever control passes out of the section of code that required the specific setting of the controls at run-time.

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 OpenVMS 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.

Finally, note that the /ROUNDING_MODE qualifier is affected in the same way as /IEEE_MODE, and is included in the whole program floating-point mode, and that because VAX floating point operations are actually performed using IEEE instructions, compilations that use VAX format floating-point exhibit the same whole program floating-point mode settings as compilations with /IEEE_MODE=DENORM/ROUND=NEAREST.

6.6.3. Predefined Macros

The compiler predefines a number of macros with the same meanings as in the native OpenVMS Alpha compiler, except that it does not predefine any of the macros that specify the OpenVMS Alpha architecture. Instead, it predefines the macros `__ia64` and `__ia64__`, as is the practice in the Intel and gcc compilers for OpenVMS 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 system. Doing this causes the CRTL headers and other OpenVMS headers to take the wrong path for OpenVMS I64. You must not define any of the OpenVMS Alpha architecture predefined macros when using this compiler.

6.7. VSI C++

VSI C++ Version 7.1 is supported on I64. Refer to the VSI C++ documentation for details and additional porting considerations.

6.7.1. Floating Point and Predefined Macros

Exactly the same considerations for floating-point defaults, /IEEE_MODE semantics, and predefined macros as described for the C compiler above apply to the C++ compiler.

6.7.2. Long Double

The long double type is always represented in 128-bit IEEE quad precision. The /L_DOUBLE_SIZE=64 qualifier is ignored with a warning. Note that on OpenVMS Alpha, C++ library support for 64-bit lon double was limited - code that requires a 64-bit floating point type should use double instead of long double.

6.7.3. Object Model

The object model and name mangling schemes used by the C++ compiler on OpenVMS I64 differ significantly from those used on OpenVMS Alpha. The "ARM" object model is not available, the only object model is one that supports the ANSI/ISO C++ standard. However, this still differs from the /MODEL=ANSI object model implemented on OpenVMS Alpha, as the model on OpenVMS I64 is an implementation of the industry-standard OpenVMS I64 ABI. Programs that rely on the layout of C ++ objects (non "POD" data), or on the external mangles names as encoded in the .obj file, will need to be reworked. Such programs are inherently highly implementation-dependent. But programs that use standard C++ features in a reasonable implementation-independent manner should not have difficulty in porting.

6.7.4. Language Dialects

The "cfront" dialect is no longer supported (and will be removed from OpenVMS Alpha as well). Compilations specifying /standard=cfront will instead use the "relaxed_ansi" dialect.

6.7.5. Templates

On I64, .OBJ files are implemented in ELF format rather than EOBJ, and along with the OpenVMS I64 linker they support the notion of "COMDAT" sections that have been used for some time on

both Windows and Unix platforms to resolve the issues of duplicate definitions at link-time that arise when using C++ templates and inline functions. On OpenVMS Alpha, these issues are handled by the repository mechanism, which arranges to present a single defining instance to the linker. On OpenVMS I64, no repository mechanism is needed to do this, as duplicates are discarded by the linker automatically. So the repository-based template instantiation options supported on OpenVMS Alpha are not supported on IPF. OpenVMS Alpha build procedures that attempt to manipulate objects in the repository will fail on OpenVMS I64 and will need to be changed (because there are no objects in the repository on OpenVMS I64, just the demangler database). In most cases, the reason for manipulating the repository directly in the build procedure has been eliminated by the compiler's use of COMDAT instantiation.

6.8. Java

Java Version 1.4.2-1 is supported on I64. There are no differences between Java on Alpha and I64.

6.9. Macro-32

Most VAX MACRO programs that compile on Alpha should recompile on 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 *VSI OpenVMS MACRO Compiler Porting and User's Guide*.

6.9.1. /ARCH and /OPTIMIZE=TUNE Qualifiers

For the sake of “compile-and-go” compatibility, OpenVMS 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.

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

6.10. VSI Pascal

VSI Pascal Version 5.9 is available for both OpenVMS Alpha and OpenVMS I64.

There are no known language differences between Pascal on Alpha and Pascal on I64. The only difference is that the default is for IEEE floating datatypes instead of VAX floating datatypes, which is true for all the VSI OpenVMS I64 compilers.

6.10.1. /ARCH and /OPTIMIZE=TUNE Qualifiers

For the sake of "compile-and-go" compatibility, OpenVMS 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.

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

Chapter 7. 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. Intel 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 Intel Itanium processor family, see the *Intel ® Itanium ® Architecture Assembly Language Reference Guide*, available at:

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

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.

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 are 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 can 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, refer to the *Alpha Architecture Handbook*.

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.

Endianism refers to the way 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, the 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 "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.

Figure 7.1. Big-Endian Byte Ordering

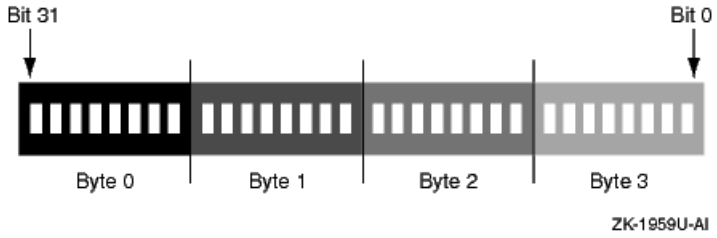
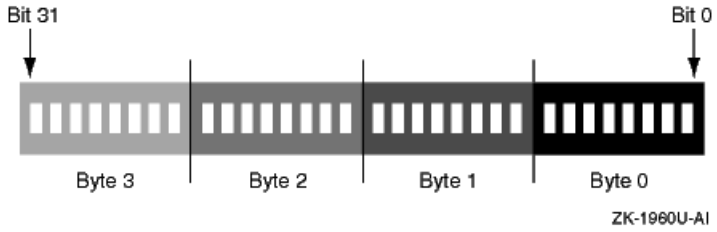


Figure 7.2. Little-Endian Byte Ordering



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 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 reverse fashion, so 0x1234 is stored as 0x12 0x34 in memory.

Consider the number 1025 (2 to the tenth power plus 1) stored in a 4-byte integer. This is represented in memory as follows:

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

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

Appendix A. Application Evaluation Checklist

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

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 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 I64. If your application is running on OpenVMS VAX, you must first port it to OpenVMS Alpha. (For more information, refer to 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:		
	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
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 I64?		
	a. No further development	[] YES	[] NO
	b. Maintenance releases only	[] YES	[] NO
	c. Additional or changed functionality	[] YES	[] NO
	d. Maintain separate I64 and OpenVMS Alpha sources	[] YES	[] NO
	[If you answer YES to a, you might want to consider translating the application. A YES response to b or c is reason to evaluate the benefits of recompiling and relinking your application, although translation is still possible. If you intend to maintain separate I64 and OpenVMS Alpha sources, as indicated by a YES to d, you might need to consider interoperability and consistency issues, especially if 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.			
	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.]		

10.	a. Do you have access to all source files that make up your application?	[] YES	[] NO
	b. If you are considering using VSI Services, will it be possible to give VSI 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 I64 images. Note that if you have OpenVMS VAX images, first translate them to OpenVMS Alpha images. You cannot translate them directly to I64 images. A YES answer to b allows you to take advantage of a greater range of VSI 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 I64. If possible, VSI 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 I64 assembly code. Similarly, PL/I, Fortran 77, and ADA 83 are not supported on I64. If your application has code written in PL/I, VSI 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 whether the newer compiler requires changes to the code. In general, if the compilers are not available on 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.]		
External Dependencies			
Consider the system configuration, environment, and software required for developing, testing, and running your application.			
	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 work loads?		
	[This will help you determine whether your entire environment is available on I64.]		
15.	Does the application rely on any special hardware?	[] YES	[] NO
	[This will help you determine whether the hardware is available on 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
	[VSI recommends having your application running on the latest version of OpenVMS Alpha before porting it to I64.]		
17.	Does the application require layered and third-party products to run?		
	a. From VSI: (other than compiler RTLs)	[] YES	[] NO
	List the VSI layered products and versions:		
	VSI Product Version		
	b. From third parties:	[] YES	[] NO
	List the third-party products and versions:		
	Third-Party Product Version		
	[If you answer YES to a and are uncertain whether the VSI layered products are yet available for I64, check with an VSI support representative. For information about VSI's commitment to support porting of software, see Section 3.3. 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 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 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 VSI support representative; see also Chapter 5. VSI recommends a disciplined testing process to check regressions in product development.]		
Dependencies on the OpenVMS Alpha Architecture			
Consider differences between OpenVMS Alpha and I64. 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-point data types?	[] YES	[] NO
	[If you answer YES, note that the default for I64 compilers is IEEE floating data types. 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, VSI 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 4.8.7 and refer to the *VSI 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$GOTO_UNWIND?	[] YES	[] NO
	[If you answer YES to a, make sure that calls to the SYS$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$GOTO_UNWIND_64. I64 does not support SYS$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 ($CRMPSC) to share data with other processes, how is data access synchronized?				
	[This will help you determine whether you will need to use explicit synchronization, and the level of effort required to guarantee synchronization among the parts of your application. Use of a high-level synchronization method generally allows you to port an application most easily.]				
23.	Does the application currently run in a multiprocessor (SMP) environment?			[] YES	[] NO
	[If you answer YES, it is likely that your application already uses adequate interprocess synchronization methods.]				
24.	Does the application use AST (asynchronous system trap) mechanisms?			[] YES	[] NO
	[If you answer YES, you should determine whether the AST and main process share access to data in process space. If so, you may need to explicitly synchronize such accesses. Note that this is equally true for applications on Alpha. For more information about ASTs, see *VSI OpenVMS Programming Concepts Manual*.]				
25.	a. Does the application contain condition handlers?			[] YES	[] NO
	b. Do any handlers modify data in the mechanism array?			[] YES	[] NO
	[The OpenVMS Alpha and OpenVMS I64 mechanism array formats differ significantly. Be sure to modify any applications that modify register values in the mechanism array. For more information, see *VSI OpenVMS Alpha Guide to Upgrading Privileged-Code Applications*.]				
26.	a. Does the application run in privileged mode or link against SYS$BASE_IMAGE?			[] YES	[] NO
	If so, why?				
	b. Does the application depend on OpenVMS internal data structures or interfaces?			[] YES	[] NO
	[Applications that link against the OpenVMS executive or run in privileged mode might require additional porting work. Undocumented interfaces in SYS$BASE_IMAGE might have changed on I64. Applications that depend on OpenVMS internal data structure definitions (as defined in SYS$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 I64.]				
27.	Does the application use connect-to-interrupt mechanisms?			[] YES	[] NO
	If yes, with what functionality?				
	[Connect-to-interrupt is not supported on I64 systems. Contact an VSI 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 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 VSI to plan performance enhancements that are most meaningful to customers.]		
32.	Does the application use any OpenVMS Alpha Calling Standard routines?	[] YES	[] NO
	[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 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 I64. For more information on the differences between OpenVMS Alpha and OpenVMS I64 calling standards, see Section 2.1.1. For more information about OpenVMS calling standards, see the *VSI 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 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 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
36.	Does the application use BLISS register Built-ins?	[] YES	[] NO
	[If you answer YES, note that applications that use the BLISS BUILTIN statement to declare register built-ins 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 available?	[] YES	[] NO
	[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 I64.]		
39.	Copy the application source modules and all related code to the I64 system. Have you copied the modules and related code to I64?	[] YES	[] NO
40.	Compile, link, and run the application on the I64 system. Application is compiled, linked, and running?	[] YES	[] NO
	[For details, see Chapter 5.]		
41.	Perform unit testing on the I64 system. Have you completed the unit testing?	[] YES	[] NO
	[For details, see Chapter 5.]		

42.	Perform system testing on the I64 system. Have you completed the system testing?	[] YES	[] NO
	[For details, see Chapter 5.]		
43.	Perform regression testing on the I64 system. Have you completed the regression testing?	[] YES	[] NO
	[For details, see Chapter 5.]		
44.	Did testing succeed?	[] YES	[] NO
	[If you answer NO, resolve the problems, then recompile, relink, and retest the application. For details, see Chapter 5.]		

Appendix B. Unsupported Layered Products

VSI has not plans to port some layered products to OpenVMS I64. Table B.1 lists these layered products and the replacement products recommended by VSI.

Table B.1. Layered Products Not Ported to OpenVMS I64

Product	Suggested Replacement
BASEstar Classic	BASEstar Open
HP Ada Compiler for OpenVMS Alpha Systems	Ada compiler from Ada Core
HP Ada Compiler for OpenVMS Alpha Systems	GNAT Ada for OpenVMS Integrity Servers – from Ada Core Technologies
Pathworks 32	Advanced Server

Appendix C. Porting Application-Specific Stack-Switching Code to I64

Note

The information in this appendix has been updated to include x86-64.

Many applications support small, private threading packages that switch stacks to perform multiple tasks within the context of a single process. Many of these packages use one or more small routines written in assembly language (Macro-32 on VAX or Macro-64 on Alpha) to manipulate the stack pointers and to cause, in effect, a context switch within a single process. Typically, the ability to stall and restart the tasks is also required. The replaced stack can be in any mode appropriate to the required task.

On I64, switching stacks is much more difficult. The I64 architecture includes two stacks, a memory stack and a register stack engine backing store (commonly called the *RSE stack* or simply the*register stack*.) The RSE stack is maintained by hardware and the architecture provides support for asynchronous manipulation of the stack. Also, the I64 architecture includes many specialized hardware registers (called control and application registers) that must be maintained across a context switch. In addition, the IAS assembler is not a supported part of the OpenVMS distribution.

To accommodate these types of applications, OpenVMS offers a set of system routines called KP services. Originally written to support device drivers written in higher-level languages, the KP services have been expanded to work in any mode and at any IPL. While the KP model may not match the needs of all private stack-switching code, VSI strongly suggests moving to this model.

KP services are available on Alpha, I64, and x86-64 architectures, allowing for common code. There are some differences in implementation; however, whenever possible, options valid on only one architecture are ignored on the other.

C.1. Overview of KP Services

The KP model is most accurately described as a co-routine model with voluntary stall and resumption. A code stream running in process or system context desires to start another stream (called the *KP routine*) in the same mode but without disturbing the existing stack context and allowing the KP routine to stall and resume as necessary while maintaining its own distinct context. Resumption of a KP routine can be asynchronous to the original code stream that started the KP routine.

The easiest view of a KP routine is that of a routine that operates on a private stack (or on I64, a pair of stacks.) The routine can be stalled and resumed. To a compiler, the routines to start, stall, and resume simply appear as outbound procedure calls which obey the OpenVMS calling standard. The saving of state and stack switch takes place entirely within the context of the KP routines. When a code stream gives up control, the resumption point is always as if the call that caused transfer of control completed.

The base support includes routines to assist in the allocation of necessary stacks and data structures and routines to start, stall, restart, and terminate a KP routine.

A KP routine can start another KP routine.

The basic KP support routines are:

- EXE$KP_START — Start a KP routine.
- EXE$KP_STALL_GENERAL — stall the current routine and return to the last caller of EXE$KP_START or EXE$KP_RESTART.
- EXE$KP_RESTART — Restart a KP routine that has been stalled by a call to EXE$KP_STALL_GENERAL.
- EXE$KP_END — Terminate the KP routine.

The routines and required data structures are described in Section C.2.2.

C.1.1. Terminology

The **memory stack** is the stack that is pointed to the stack pointer register. The memory stack is the only stack on Alpha and x86-64 systems.

The **register stack** is the informal name for the **register stack engine (RSE) backing store**. The I64 architecture includes 96 stacked registers that are used for argument passing between routines and that can be used for local storage within a routine as well. The hardware maintains the stacked register state and flushes registers to the backing store as needed.

A **KP routine** is a sequence of code that runs on a private stack or stacks. It is started via a call to EXE$KP_START and is terminated by an implicit or explicit call to EXE$KP_END.

A **KPB** is the data structure that describes the stack or stacks and that maintains state and context information for the KP routine.

A KPB is **valid** if it has been used to activate a KP routine via EXE$KP_START. EXE$KP_END marks the KPB as **invalid**. The initial state of a KPB is invalid.

A KPB is **active** when the KP routine has been started via EXE$KP_START or resumed via EXE$KP_RESTART. EXE$KP_STALL_GENERAL and EXE$KP_END mark the KPB as **inactive**. The initial state of a KPB is inactive.

C.1.2. Stacks and Data Structures

On I64, three pieces of memory are associated with a KP routine — a memory stack, an RSE stack, and a KPB, which is the data structure that ties them all together.

Note

On non-I64 architectures, there is no RSE stack and all parameters and fields related to the RSE stack are ignored.

The KPB and stacks each must be allocated from an appropriate region, and at an appropriate mode, protection, and ownership to match the mode in which the KP routine will execute. When porting an existing application, it is expected that the application already allocates an appropriate memory stack. The existing memory stack allocation routine can be adapted to the KP API. As with previous architectures, the memory stack is accessed from the highest address (the base of the stack) to the lowest address.

RSE stacks are typically allocated from 64-bit space because the register stack engine is a new entity with no previous 32-bit dependencies. A number of allocation routines have been supplied that should cover most common application needs. The RSE stack is accessed from the lowest address to the highest address.

Table C.1 offers guidelines for allocation by mode and scope of the application.

Table C.1. Allocation Guidelines by Mode and Scope

Mode-Scope[1]	**KPB**	**Memory Stack**	**Register Stack**
Kernel – System EXE$KP_ ALLOC_KPB[2]	Nonpaged pool KW EXE$ALO NONPAGED	S0/S1 KW EXE$KP_ALLOC_ MEM_STACK	S2 KW EXE$KP_ALLOC_ RSE_STACK[3]
Kernel – Process	Non-Paged Pool or P1 KW EXE$ALO NONPAGED or EXE$ALOP1PROC	P1 – Permanent KW $CREATE_REGION /$CRETVA	P2 – Permanent KW EXE$KP_ALLOC_ RSE_STACK_P2
Kernel – Image	P1 KW EXE$ALOP1IMAG	P1-Non-permanent KW $CREATE_REGION /$CRETVA	P2 – Non-permanent KW $CREATE_REGION /$CRETVA
Exec – Process	P1 EW EXE$ALOP1PROC	P1 – Permanent EW $CREATE_REGION /$CRETVA	P2 – Permanent EW EXE$KP_ALLOC_ RSE_STACK_P2
Exec – Image	P1 EW EXE$ALOP1IMAG	P1 – Non-permanent EW $CREATE_REGION /$CRETVA	P2 – Non-permanent EW $CREATE_REGION /$CRETVA
Super – Process	P1 SW EXE$ALOP1PROC	P1 – Permanent SW $CREATE_REGION /$CRETVA	P2 – Permanent SW EXE$KP_ALLOC_ RSE_STACK_P2
Super – Image	P1 SW EXE$ALOP1IMAG	P1 – Non-permanent SW $CREATE_REGION	P2 – Non-permanent SW $CREATE_REGION

Mode-Scope[1]	KPB	Memory Stack	Register Stack
		/$CRETVA	/$CRETVA
User – Image	P0 UW Heap/ Malloc/LIB$GET_VM	P0 – Non-permanent[4] UW EXE$KP_ALLOC_ MEM_STACK_USER	P2 – Non-permanent UW EXE$KP_ALLOC_ RSE_STACK_P2

[1]Image scope terminates at image exit. Process scope terminates at process exit and will survive image rundown. System scope does not require process context.

[2]EXE$KP_ALLOC_KPB allocates kernel mode KPB and kernel mode RSE stacks in a single call

[3]EXE$KP_ALLOC_RSE_STACK_P2 creates permanent regions.

[4]Note that permanent memory regions may not be created in user mode.

C.1.3. KPBs

The KPB is a data structure that is used to maintain the necessary context between the initiating code stream and the KP routine. The KPB is semitransparent. Some fields are maintained by the application, some by the KP routines and some are shared. The KP routines assume the KPB was zeroed on allocation and, thus that any nonzero field contains legitimate data.

The structure definitions for a KPB are defined by the $KPBDEF macro for Macro-32 and KPBDEF.H for C. The KPB definitions are considered system-internal and thus supplied in LIB.MLB and SYS$LIB_C.TLB. For BLISS, LIB.REQ or LIB.L32/LIB.L64 contain the KPB definitions.

The KPB is a variable-length structure consisting of a number of areas or substructures. Not all areas are required. The areas are:

- Base area
- Scheduling area
- VEST area
- Spinlock area
- Debug area
- User parameter area

The base area is required. It contains a standard structure header, the stack sizes and base addresses, flags (including what other areas are present), the memory stack pointer for the nonactive code stream, pointers to the other areas, and additional fields required by the base KP routines.

The scheduling area includes pointers to stall, restart and end handling routines, a fork block and a pointer to an additional fork block. With the exception of the end routine, most are required by high-IPL driver-level code only. Callers of EXE$KP_USER_ALLOC_KPB must supply an end routine to perform necessary cleanup of the allocated memory.

The VEST and spinlock areas are used primarily by the driver code.

The debug area offers limited tracing capability implemented in the driver support routines.

The user parameter area is simply undefined storage allocated contiguously to the other areas. The application is free to use this memory for its own needs.

C.1.4. Supplied KPB Allocation Routines

The operating system supplies two standard allocation routines for KPBs with associated stacks. The original kernel-mode, driver level interface has been retained unchanged from Alpha so that device drivers using the KP interface do not require source changes in this area. In addition, a mode-independent routine is available. The mode-independent routine calls application-specified routines to allocate the KPB and each of the stacks. Most new applications and applications porting to the KP API will use the latter routine.

Both the kernel and mode-independent routines initialize the KPB. C prototypes for all the supplied routines can be found in the header file EXE_ROUTINES.H.

C.1.5. Kernel Mode Allocation

The format for kernel mode allocation is as follows.

```
EXE$KP_ALLOCATE_KPB kpb, stack_size, flags, param_size
```

C prototype

```
status = EXE$KP_ALLOCATE_KPB( KPB_PPS kpb,
    int stack_size,
    int flags,
    int param_size)
```

For kernel mode use only. This routine has the same prototype as the original Alpha routine.

On I64, the RSE stack size equals the memory stack size (in bytes).

Note

On non-I64 architectures, there is no RSE stack and all parameters and fields related to the RSE stack are ignored.

Parameters

- **KPB –** Address of a longword to receive the address of the allocated data structure. By convention within OpenVMS APIs, 32-bit pointers are passed by 32-bit address (short pointer to short pointer to struct KPB).
- **STACK_SIZE –** A 32-bit value denoting the size of the stacks to be allocated in bytes. Passed by value. The supplied value is rounded up to a multiple of the hardware page size. In any case, the minimum number of pages allocated will be no smaller than the SYSGEN parameter KSTACKPAGES. The stack is allocated on page boundaries with unmapped guard pages on both ends. The memory stack is allocated in 32-bit S0/S1 address space.

  ### Note

  On I64 systems, the RSE stack is sized identically to the memory stack and is allocated from 64-bit S2 address space.

- **FLAGS –** Longword bitmask of flags. Passed by value. Table C.2 describes the flags that are defined for this parameter.

Table C.2. Kernel Mode Allocation Flags

Flag	Description
KP$M_VEST	OpenVMS system KPB. In general, this flag should be set.
KP$M_SPLOCK	Allocate a spinlock area within the KPB
KP$M_DEBUG	Allocate a debug area within the KPB.
KP$M_DEALLOC_AT_END	KPB should be automatically deallocated when the kernel process routine terminates.
KP$M_SAVE_FP (IA64 only; ignored on Alpha and x86-64)	Save floating-point context as well as the general registers. Certain operations, such as integer multiplication and division on IA64, can be implemented using floating-point operations. These operations use the minimal floating-point register set, which is by definition not part of the preserved register set. If the application uses only the minimal floating-point register set, this bit need not be set. If the application uses floating-point data, this bit must be set to preserve the correct floating point context across the stack switch. In the x86-64 calling standard, no floating-point registers are preserved across procedure calls.
KP$M_SET_STACK_LIMITS	Call $SETSTK_64 at every stack switch. Process-scope applications should always set this flag because condition handling requires accurate stack limit values.

- **PARAM_SIZE –** Longword by value. Size, in bytes, of the user parameter area in the KPB. Pass zero if no parameter area is required.

Return value (status)

SS$_NORMAL
SS$_INSFMEM
SS$_INSFARG
SS$_INSFRPGS

C.1.6. Mode-Independent Allocation

The syntax for mode-independent allocation is as follows.

```
EXE$KP_USER_ALLOC_KPB kpb, flags, param_size, *kpb_alloc, mem_stack_bytes,
*memstk_alloc, rse_stack_bytes, *rsestk_alloc, *end_rtn
```

C prototype

```
status = EXE$KP_USER_ALLOC_KPB(  KPB_PPS kpb, int flags,
     int param_size,
```

```
int (*kpb_alloc)(),
int mem_stack_bytes,
int(*memstk_alloc)(),
int rse_stack_bytes,
int(*rsestk_alloc)(),
void(*end_rtn)())
```

Parameters

- **KPB –** Address of a longword to receive the address of the allocated data structure. By convention within OpenVMS APIs, 32-bit pointers are passed by 32-bit address (short pointer to short pointer to struct KPB).
- **FLAGS –** Longword bitmask of flags. Passed by value. Table C.3 describes the flags that are defined for this parameter.

Table C.3. Mode-Independent Allocation Flags

Flag	Description
KP$M_VEST	OpenVMS system KPB. In general, this flag should be set.
KP$M_SPLOCK	Allocate a spinlock area within the KPB.
KP$M_DEBUG	Allocate a debug area within the KPB.
KP$M_DEALLOC_AT_END	KPB should be automatically deallocated when the kernel process routine terminates.
KP$M_SAVE_FP (IA64 only; ignored on Alpha and x86-64)	Save floating-point context as well as the general registers. Certain operations such as integer multiplication and division on I64 systems can be implemented using floating-point operations. These operations use the minimal floating-point register set, which is by definition not part of the preserved register set. If the application uses only the minimal floating-point register set, this bit need not be set. If the application uses floating-point data, this bit must be set to preserve the correct floating-point context across the stack switch. In the x86-64 calling standard, no floating-point registers are preserved across procedure calls.
KP$M_SET_STACK_LIMITS	Call $SETSTK_64 at every stack switch. Process-scope applications should always set this flag since condition handling requires accurate stack limit values.

- **PARAM_SIZE –** Longword by value. Size, in bytes, of the user parameter area in the KPB. Pass zero if no parameter area is required.
- **KPB_ALLOC –** Address of a procedure descriptor for the routine to allocate the KPB. The allocation routine will be called with two parameters, a length and a longword to receive the address of the allocated KPB. The length parameter is in bytes and is passed by reference. The allocation routine must allocate at least the required number of bytes out of 32-bit space appropriate for the

mode in which the KP routine will run. The address is expected to be at least quadword aligned and the actual allocation size must be written back into the length argument.

- **MEM_STACK_BYTES –** 32-bit value denoting the size of the memory stack to be allocated in bytes. Passed by value. The supplied value is rounded up to a multiple of the hardware page size.
- **MEMSTK_ALLOC –** Address of a procedure descriptor for a routine to allocate the memory stack. Section C.1.7 describes the format and required actions of this routine.
- **RSE_STACK_BYTES –** 32-bit value that denotes the size, in bytes, of the memory stack to be allocated. Passed by value. The supplied value is rounded up to a multiple of the hardware page size. On non-I64 architectures, this parameter is ignored.
- **RSESTK_ALLOC –** Address of a procedure descriptor for a routine to allocate the memory stack. Section C.1.7 describes the format and required actions of this routine. On non-I64 architectures, this parameter is ignored.
- **END_RTN –** Address of a procedure descriptor for the end routine. The end routine is called when the KP routine terminates, either by calling EXE$KP_END or by returning. An end routine is required to either cache or deallocate the stacks and KPB.

C.1.7. Stack Allocation APIs

The stack allocation routines have identical APIs for both memory and RSE stack allocation. The routine is called with a 64-bit address of the allocated KPB and an integral number of hardware-specific pages (not pagelets) to allocate.

The syntax for specifying stack allocation routines is as follows:

```
status = alloc-routine (KPB_PQ kpb, const int stack_pages)
```

- **KPB –** 64-bit address of a previously allocated KPB. Passed by 64-bit reference.
- **STACK_PAGES –** Integral number of whole pages to allocate. Passed by 32-bit value.

The allocation routine is expected to allocate page-aligned address space. While not strictly necessary, it is strongly suggested that the stack be protected by no-access guard pages on both ends. VSI also recommends that the minimum stack size be at least the value of the SYSGEN parameter KSTACKPAGES (global cell SGN$GL_KSTACKPAG). This allows a certain measure of control of the stack size without necessitating recompilation of the application. Also, stack usage on I64 is significantly different than on previous architectures, and the previously allocated stack size might not be adequate.

The memory stack allocation routine must set the following KPB fields as follows:

- KPB$IS_STACK_SIZE – The size of the stack in bytes, not including guard pages.
- KPB$PQ_STACK_BASE – The address of the stack base. For the memory stack, this is the address of the byte past the end of the allocated stack. In other words, this is the sum of the allocated address plus the size of the allocation in bytes.

The memory stack allocation routine can set the following KPB field as follows:

- KPB$Q_MEM_REGION_ID – The region ID returned by the $CREATE_REGION system service. This information is required to deallocate the stack.

The RSE stack allocation routine must set the following KPB fields as follows:

- KPB$IS_STACK_SIZE – The size of the stack in bytes, not including guard pages.
- KPB$PQ_STACK_BASE – The address of the stack base. For the RSE stack, this is the lowest allocated address.

The RSE stack allocation routine can set the following KPB field as follows:

- KPB$Q_REGION_ID – The region ID returned by the $CREATE_REGION system service. This information is required to deallocate the stack.

Both routines must return status to the calling routine.

C.1.8. System-Supplied Allocation and Deallocation Routines

The system supplies a number of standard allocation routines. The routines adhere to the KP allocation API and can be used in place of a user-written routine if they meet the needs of the application. The following allocation routines are available:

- EXE$KP_ALLOC_MEM_STACK (kernel mode, S0/S1 space)
- EXE$KP_ALLOC_MEM_STACK_USER (user mode, P0 space)
- EXE$KP_ALLOC_RSE_STACK (caller's mode, S2 space)
- EXE$KP_ALLOC_RSE_STACK_P2 (caller's mode, P2 space)

The following deallocation routines are supplied. All deallocation routines take a single argument, the address of the KPB.

- EXE$KP_DEALLOCATE_KPB
  - Only KPBs allocated by EXE$KP_ALLOCATE_KPB
  - Deallocates stacks and KPB
- EXE$KP_DEALLOC_MEM_STACK
- EXE$KP_DEALLOC_MEM_STACK_USER
- EXE$KP_DEALLOC_RSE_STACK
- EXE$KP_DEALLOC_RSE_STACK_P2

C.1.9. End Routine

The end routine is called by the KP services when EXE$KP_END is called, either explicitly or by reaching the end of the KP routine. Since EXE$KP_USER_ALLOC_KPB allows the specification of arbitrary allocation routines, the end routine must either cache the KPB for future use by the application or call the necessary deallocation routines for the stacks and KPB.

The end routine is called with two parameters as follows:

```
void end_routine (KPB_PQ KPB, int status)
```

- **KPB –** 64-bit address of a previously allocated KPB. Passed by 64-bit reference.
- **STATUS –** optional status value. If EXE$KP_END was called explicitly, the status value is that supplied as the second argument to EXE$KP_END. If the second argument was omitted, the value SS$_NORMAL is supplied. If the KP routine terminated by returning, the value supplied is the return value of the KP routine.

C.2. KP Control Routines

Once the KPB and stacks are allocated, the four routines that determine the state of a KP routine can be used.

C.2.1. Overview

A KP routine is initiated by a call to EXE$KP_START. While running, the KP routine may elect to give up control by calling EXE$KP_STALL_GENERAL. A stalled KP routine can be resumed by calling EXE$KP_RESTART. The KP routine is terminated either by an explicit call to EXE$KP_END or by returning from the routine. In the latter case, the KP services call EXE$KP_END implicitly.

When a KP routine starts, the current thread of execution state is saved onto the current stack, the KP stack is loaded, and the KP routine is called.

When the KP routine stalls, the KP routine's context is saved onto the KP stack, the stack is switched to the original stack, and the main routine's context is loaded from stack. The result is that a stall causes the original routine to return from the last call to EXE$KP_START or to EXE$KP_RESTART.

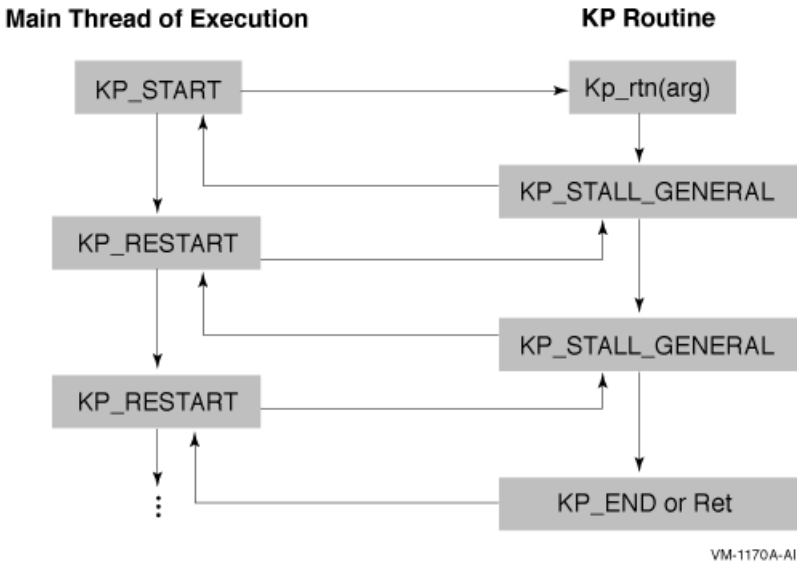
When the KP routine is restarted, the current context is saved onto the current stack, the stack is switched to the KP routine's stack and the KP routine's context is restored from the stack. The KP routine returns from the last call to EXE$KP_STALL_GENERAL.

The stall/resume sequence can occur zero or more times. The KP routine is not required ever to stall. A stalled routine cannot stall again until it has been resumed. A running KP routine cannot be restarted until it has stalled. The full checking version of SYSTEM_PRIMITIVES.EXE enforces these rules. Failure to follow these rules can result in a KP_INCONSTATE bugcheck, depending on the mode in which the KP routine is running.

When the KP routine terminates, either by explicitly calling EXE$KP_END or by returning to the caller, no current context is saved, the stack is switched, and the original thread context is restored. At this point, if the DEALLOCATE_AT_END flag is set (kernel mode only) or if an end routine address has been supplied, the appropriate action takes place. The original thread returns from the call that started or restarted the KP routine.

Figure C.1 shows the overall code flow.

Figure C.1. KP Routine Execution



Note

While the main thread of execution is depicted as a continuous stream in the above diagram, the actual thread of execution may include asynchronous components. The application is required to perform all necessary synchronization as well as maintaining the necessary scope of the KPB.

C.2.2. Routine Descriptions

This section describes the routines.

C.2.2.1. EXE$KP_START

Syntax:

```
status = EXE$KP_START(kpb, routine, reg-mask)
```

- **kpb –** Address of a previously allocated and initialized KPB. Passed by 32-bit reference.
- **routine –** KP routine address
- **reg-mask –** Register save mask. Longword. Passed by value. This argument is read on Alpha only. The I64 and x86-64 interfaces supports only the OpenVMS Calling Standard for register preservation. The constant KPREG$K_HLL_REG_MASK, defined in KPBDEF, can be used for calls from higher-level languages on Alpha to specify the calling standard register mask.

This routine suspends the current thread of execution, swaps to the new stacks; and calls the specified routine. The KP routine is called with a single argument – the 32-bit address of the supplied KPB. The KPB must be invalid and inactive.

C.2.2.2. EXE$KP_STALL_GENERAL

Syntax:

```
status = EXE$KP_STALL_GENERAL(kpb)
```

- **kpb –** Address of the KPB passed at the start of this routine. Passed by 32-bit reference.

This routine stalls the current thread of execution, saving context onto the KP stack, and returns to the most recent call that started or restarted this routine. The KPB must be valid and active.

The return status from this routine is supplied by the routine that restarts this procedure.

C.2.2.3. EXE$KP_RESTART

Syntax:

```
EXE$KP_RESTART(kpb [, thread_status])
```

- **kpb –** Address of the KPB last used to stall this routine. Passed by 32-bit reference.
- **thread_status –** Status value to be supplied as the return value for the call to EXE$KP_STALL_GENERAL that last stalled the KP routine. Passed by value. Optional. If this parameter is omitted, SS$_NORMAL is returned.

This routine causes the stalled KP routine to restart by returning from the last call to EXE$KP_STALL_GENERAL. Note that this may be a completely asynchronous operation with respect to the original thread of execution that started the KP routine. The KPB must be valid and inactive.

C.2.2.4. EXE$KP_END

Syntax:

```
status = EXE$KP_END(kpb [, status])
```

- **kpb –** Address of the KPB last used to start or restart this routine. Passed by 32-bit reference.
- **status –** Status value to be supplied to the end routine, if any was specified in the KPB. Passed by value. Optional. If this parameter is omitted, SS$_NORMAL is returned.

This routine terminates the KP routine, returning control to the last thread of execution that started or restarted the KP routine. The KPB must be valid and active. The KPB is marked as invalid and inactive and cannot be used in subsequent calls to EXE$KP_RESTART or EXE$KP_STALL_GENERAL without first calling EXE$KP_START to start another KP routine.

Instead of calling EXE$KP_END, the KP routine can return to the caller. Returning to the caller causes the KP code to call KP_END automatically. In that case, the return status from the KP procedure is used for the status argument.

C.3. Design Considerations

- KP routines run in a single process. The KP services by themselves offer no parallelism benefits on multiple-CPU systems.
- All calls to EXE$KP_STALL_GENERAL, EXE$KP_RESTART and EXE$KP_END must occur in the same mode as the call to EXE$KP_START. Mode changes can occur between calls, but all calls must be made at the same mode.
- Multiple KPBs can be valid at the same time.
- Multiple KPBs can be active at the same time. This implies that a KP routine has started or restarted another KPB. Except in the simplest cases, code flow rapidly becomes exceedingly complex as the

number of active KPBs increases. In such cases, a work queue model with a dispatcher often removes the need for multiple active KPBs.

- Allocation and deallocation of stacks is sufficiently expensive in system time and resources that consideration should be given to some level of caching KPBs within the application instead.
- Applications using KP services in process context in kernel mode must lock the stacks into the working set.