HP OpenVMS

Migrating an Application from OpenVMS VAX to OpenVMS I64

May 2006

This manual describes how to create an HP OpenVMS Industry Standard 64 for Integrity Servers (I64) version of an OpenVMS VAX application.

Revision/Update Information:	This is a new manual.
Software Version:	OpenVMS I64 Version 8.2 and higher OpenVMS VAX Version 6.1 and higher

Hewlett-Packard Company
Palo Alto, California

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

Contents

3 Migrating Your Application

4 Overview of Recompiling and Relinking

5 Adapting Applications to a Larger Page Size

6 Preserving the Integrity of Shared Data

7 Checking the Portability of Application Data Declarations

8 Examining the Condition-Handling Code in Your Application

9 OpenVMS I64 Compilers

Examples

Figures

Tables

Preface

This manual is designed to assist developers in moving OpenVMS VAX applications to an OpenVMS I64 system or a mixed-architecture cluster.

Intended Audience

This manual is intended for experienced software engineers responsible for migrating application code written in high- or mid-level programming languages.

Document Structure

The manual consists of the following chapters:

- Chapter 1 provides an overview of the relationship of OpenVMS and the VAX and Itanium architectures, and of the process of migrating an application from a VAX to an I64 system. It includes information about the following:
  - Areas in which OpenVMS I64 is highly compatible with OpenVMS VAX
  - Comparison of the Intel® Itanium® architecture with the VAX architecture
  - Overview of the stages in the migration process
  - The two main migration paths—recompiling source code and translating VAX images
  - Migration support available from HP
- Chapter 2 considers the differences between the two main migration paths and the issues involved in choosing which path to take in migrating your application. It also describes how to analyze the individual parts of your application to identify architectural differences that affect migration and how to assess what is involved in resolving those differences.
- Chapter 3 describes the steps in the actual migration, from setting up your migration environment to integrating the migrated application into a new environment.
- Chapter 4 provides an overview of converting your application by recompiling and relinking.
- Chapter 5 describes how to handle dependencies your application has on the VAX page size.
- Chapter 6 describes how to handle dependencies your application has on the synchronization provided by the VAX architecture with regard to data access by multiple processes.
- Chapter 7 describes the implications of data declarations on an I64 system, including alignment concerns.

- Chapter 8 describes how to handle dependencies your application has on the VAX condition-handling facility.
- Chapter 9 contains brief summaries of the new and changed features supported by the Ada, C, COBOL, Fortran, and Pascal programming languages on OpenVMS I64 systems.
- Appendix A contains a checklist that you can use to evaluate your application for migration from OpenVMS VAX to OpenVMS I64.

Related Documents

A number of archived OpenVMS manuals describe various aspects of the porting process. These manuals are available from "Porting Documentation" navigation bar on the following Web location:

`http://h71000.www7.hp.com/doc/`

These manuals includes the following:

- *OpenVMS Migration Software for VAX to Alpha Systems: Translating Images* describes the VAX Environment Software Translator (VEST) utility. This manual is distributed with the optional layered product, DECmigrate for OpenVMS Alpha, which supports the migration of OpenVMS VAX applications to OpenVMS Alpha systems. The manual describes how to use VEST to convert most user-mode VAX images to translated images that can run on Alpha systems; how to improve the run-time performance of translated images; how to use VEST to trace Alpha incompatibilities in a VAX image back to the original source files; and how to use VEST to support compatibility among native and translated run-time libraries. The manual also includes complete VEST command reference information.
- *HP OpenVMS Migration Software for Alpha to Integrity Servers: Guide to Translating Images* describes how to use the HP OpenVMS Migration Software for Alpha to Integrity Servers (OSMAI) to migrate OpenVMS Alpha applications to OpenVMS I64.
- *Creating an OpenVMS AXP Step 2 Device Driver From an OpenVMS VAX Device Driver* describes how to convert an OpenVMS VAX device driver to run on an OpenVMS Alpha system. The book identifies the specific changes required to prepare an OpenVMS VAX device driver to be compiled, linked, loaded, and run as an OpenVMS Alpha device driver. It also contains reference material about the entry points, system routines, data structures, and macros used in OpenVMS I64 Alpha drivers.

The "OpenVMS Floating-Point Arithmetic on the Intel® Itanium® Architecture" white paper describes how floating-point data types on OpenVMS I64 differ from other platforms. You can obtain the white paper from the following location:

`http://h71000.www7.hp.com/openvms/integrity/resources.html`

The HP OpenVMS Migration Software for Alpha to Integrity Servers comprises a set of tools for translating and porting OpenVMS Alpha applications to OpenVMS I64 systems. For more information, including a link to the product documentation, see the following location:

`http://71000.www7.hp.com/openvms/products/omsva/osmais.html`

In addition, the following general programming manuals contain current information on issues discussed here:

- *VAX Architecture Reference Manual*
- *HP OpenVMS Guide to Upgrading Privileged-Code Applications*
- *VAX/VMS Internals and Data Structures*
- *HP OpenVMS Programming Concepts Manual*
- *OpenVMS Programming Interfaces: Calling a System Routine*
- *Guide to the POSIX Threads Library*
- *HP OpenVMS Calling Standard*
- *HP OpenVMS Debugger Manual*
- *HP OpenVMS Linker Utility Manual*
- *HP OpenVMS System Analysis Tools Manual*

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

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

`http://www.hp.com/go/openvms`

Reader's Comments

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

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

How to Order Additional Documentation

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

`http://www.hp.com/go/openvms/doc/order`

Conventions

The following conventions may be used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.

Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.) In the HTML version of this document, this convention appears as brackets, rather than a box.
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
|	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold type	Bold type represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
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.

1

Overview of the Migration Process

For many applications, migrating from OpenVMS VAX to OpenVMS I64 is straightforward. If your application runs only in user mode and is written in a standard high-level language, you most likely can recompile it with a native OpenVMS I64 compiler and relink it to produce a version that runs successfully on an OpenVMS I64 system. This book is intended to help you evaluate your application and to handle the relatively few cases that are more complicated.

1.1 Compatibility of VAX and I64 Systems

The OpenVMS I64 operating system is designed to preserve as much compatibility with the OpenVMS VAX user, system management, and programming environments as possible. For general users and system managers, OpenVMS I64 has the same interfaces as OpenVMS VAX. For programmers, the goal is to come as close as possible to a "recompile, relink, and run" model for migration.

Many aspects of an application running on an OpenVMS VAX system remain unchanged on an I64 system.

User Interface

- **DIGITAL Command Language (DCL)**

  The DIGITAL Command Language (DCL), the standard user interface to OpenVMS, remains essentially unchanged with OpenVMS I64. All commands, qualifiers, and lexical functions available on OpenVMS VAX also work on OpenVMS I64.

- **Command Procedures**

  Command procedures written for earlier versions of OpenVMS VAX continue to work on an OpenVMS I64 system without change. However, certain command procedures, such as build procedures, must be changed to accommodate new compiler qualifiers and linker switches. Linker options files also require modification, especially for shareable images.

- **DECwindows**

  The window interface, DECwindows Motif, is unchanged.

- **DECforms**

  The DECforms interface is unchanged.

- **Editors**

  The two standard OpenVMS editors, EVE and EDT, are unchanged.

System Management Interface

The system management utilities are mostly unchanged. One major exception is that device configuration functions, which appear in the System Generation utility (SYSGEN) on VAX systems, are provided in the System Management utility (SYSMAN) for OpenVMS I64.

Programming Interface

In general, the system service and run-time library (RTL) calling interfaces remain unchanged. You do not need to change the definitions of arguments. The few differences fall into two categories:

- Some system services and RTL routines (such as the memory management system and exception-handling services) operate somewhat differently on VAX and OpenVMS I64 systems. See the *HP OpenVMS System Services Reference Manual* and the *HP OpenVMS RTL Library (LIB$) Manual* for further information.
- A few RTL routines are so closely tied to the VAX architecture that their presence on an OpenVMS I64 system is not meaningful:

Routine Name	Restriction
LIB$DECODE_FAULT	Decodes VAX instructions.
LIB$DEC_OVER	Applies to VAX Processor Status Longword (PSL) only.
LIB$ESTABLISH	Similar functionality supported by compilers on OpenVMS I64 systems.
LIB$FIXUP_FLT	Applies to VAX PSL only.
LIB$FLT_UNDER	Applies to VAX PSL only.
LIB$INT_OVER	Applies to VAX PSL only.
LIB$REVERT	Supported by compilers on OpenVMS I64 systems.
LIB$SIM_TRAP	Applies to VAX code.
LIB$TPARSE	Requires action routine interface changes. Replaced by LIB$TABLE_PARSE.

Most VAX images that call these services and routines work when translated and run under the Translated Image Environment (TIE) on OpenVMS I64. For more information about TIE, see *OpenVMS Migration Software for VAX to Alpha Systems: Translating Images*.

Data

- The on-disk format for ODS-2 data files is the same on VAX and Integrity server systems. However, ODS-1 files are not supported on OpenVMS I64.
- Record Management Services (RMS) and file management interfaces are unchanged.
- The IEEE little-endian data types S_floating and T_floating have been added.
- Most VAX data types are supported by compilers on OpenVMS I64. For more information, see Section 2.6.2.

Databases

Standard HP databases function the same on VAX and OpenVMS I64 systems.

Network Interfaces

VAX and OpenVMS I64 systems both support the following networking interfaces:

- Interconnects
  - Ethernet
  - X.25
- Protocols
  - DECnet (Phase IV in Version 8.2; Phase V in the optional DECnet-Plus kit)
  - TCP/IP
  - OSI
  - LAD/LAST
  - LAT (Local Area Transport)
- Peripheral connections
  - SCSI
  - Ethernet
  - PCI

1.2 User-Written Device Drivers

There is no method for porting an OpenVMS VAX device driver directly to OpenVMS I64. HP recommends that you first port your OpenVMS VAX device driver to OpenVMS Alpha. After that, the port to OpenVMS I64 should be straightforward.

Porting a driver from VAX to Alpha is covered in the archived manual *Creating an OpenVMS AXP Step 2 Device Driver From an OpenVMS VAX Device Driver*. See the "Related Documents" section in the Preface of this manual for the Web location of that document. After following the procedures described in that device driver manual, you might need to make further modifications to your device drivers that are relevant changes in the OpenVMS kernel when 64-bit support was introduced in OpenVMS Alpha V7.0. For more information, see *HP OpenVMS Guide to Upgrading Privileged-Code Applications*.

Be sure to observe the following recommendations when porting device drivers from OpenVMS VAX to OpenVMS Alpha:

- Refrain from explicit use of any register numbered higher than the AP (R12). This will improve the likelihood of a simple recompile for Alpha Macro-32 code.
- Be careful if your device driver has any assumption that a PFN is only 32-bits wide, which is the limit on VAX and Alpha. OpenVMS I64 supports the 50-bit physical addresses that are supported by HP Integrity servers. As a result, the PFN field requires more than 32 bits of the 64-bit PTE.

For more information about PFNs, see *HP OpenVMS Guide to Upgrading Privileged-Code Applications*.

- An OpenVMS VAX driver for some VAX option cards might have specific references to bus support. For example, if you have a VAX Q-bus adapter, you need to find an equivalent PCI adapter and then rewrite your driver.
- If you have a "class driver" or other similar driver, usually loaded with the /NOADAPTER switch, it is not tied to a particular bus option card. This is a better candidate for porting to I64.
- If your driver will run on both Version 7.*x* and Version 8.*x* versions of OpenVMS, the driver must be compiled and linked for both versions because of internal data structure changes made for Version 8.2. For example, a driver linked on Version 8.2 will not run on Version 7.3-2 and visa versa.

Many OpenVMS VAX device drivers are written in VAX Macro-32. As such, if the driver is not overly large, it might be worth considering rewriting it, in whole or in part, in C.

For more information about writing new OpenVMS Alpha device drivers, refer to *Writing OpenVMS Alpha Device Drivers in C*.

1.3 Migration Process

The process for converting your VAX programs to run on an I64 system includes the following stages:

1. Evaluate the code to be migrated:
   - Take inventory of the elements of your application and its environment. Identify any dependencies on other programs.
   - Review code in each element to find potential obstacles to migration.
   - Decide on the best method for moving each part of the application to the I64 system.
2. Write a migration plan.
3. Set up the migration environment.
4. Migrate your application.
5. Debug and test the migrated application.
6. Integrate the migrated software into a software system.

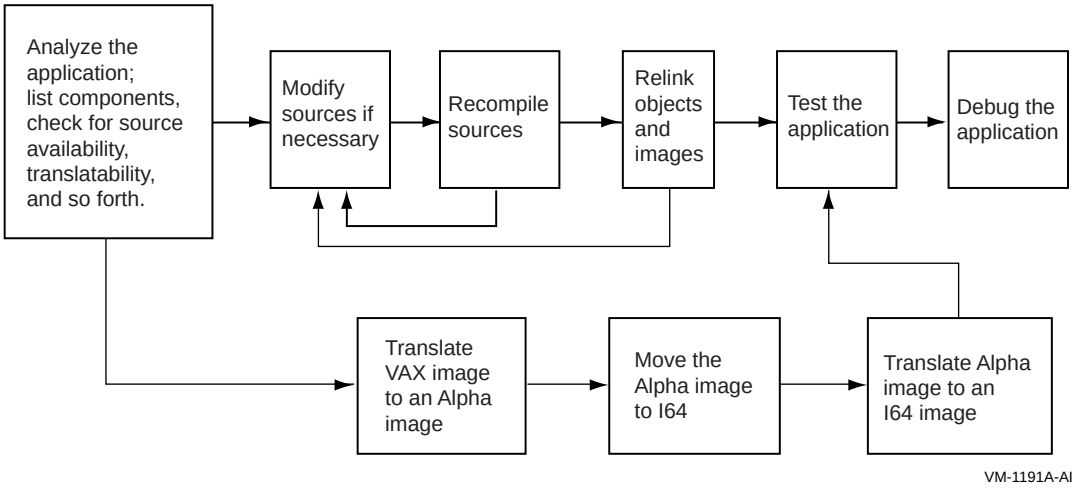
A number of tools and HP services are available to help you migrate your applications to OpenVMS I64. These tools are described in the context of the process described in this manual. The migration services are summarized in Section 1.5.

1.4 Migration Paths

There are two ways to convert a program to run on an I64 system:

- Recompiling and relinking, which creates native I64 images
- Translating, which creates native I64 images with some routines emulated under TIE

Figure 1–1 Methods for Moving VAX Applications to an I64 System



These two methods are shown in Figure 1–1. Section 2.2 discusses factors to consider when choosing a migration method.

Recompiling and Relinking

The most effective way to convert a program from OpenVMS VAX to OpenVMS I64 is to recompile the source code using a native I64 compiler (such as C or Fortran), and then use the OpenVMS Linker to relink the resulting object files and any required shareable images. This method produces a native I64 image that takes full advantage of the speed of the Integrity system.

Translating

In order to translate an image from VAX to I64, you must first translate it to Alpha using the VEST utility, then translate the Alpha image to I64.

The translation process provides a higher degree of VAX compatibility, but because the translated image does not provide the same high performance as a recompiled image, translation is used primarily as a safety net when recompiling is impossible or impractical. For example, translation is appropriate in the following situations:

- When an appropriate compiler is not available for the target system
- When source files are not available

For additional information, see Section 2.4.

1.5 Migration Support from HP

HP offers a variety of services to help you migrate your applications to OpenVMS I64.

HP customizes the level of service to meet your needs. The VAX-to-Integrity migration services available include the following:

- Migration Assessment
- Application Migration Detailed Analysis and Design

- System Migration Detailed Analysis and Design
- Application Migration
- System Migration

To determine which services are appropriate for you, contact an HP support representative or authorized reseller.

1.5.1 Migration Assessment Service

The Migration Assessment service assesses the VAX system and application environment to be migrated to the Integrity platform. The objectives of the migration are reviewed and a complete current state configuration is completed. The desired end state is determined and risks and constraints are identified. Finally, several migration scenarios are developed.

1.5.2 Application Migration Detailed Analysis and Design Service

The Application Migration Detailed Analysis and Design service provides a detailed analysis of an in-house developed application, creates a report of all VAX dependencies within all modules, and makes recommendations as to what modifications are required to migrate the application to I64. Acceptance criteria is specified for performance and functionality.

1.5.3 System Migration Detailed Analysis and Design Service

The System Migration Detailed Analysis and Design service performs a detailed analysis of the current system environment, which includes hardware, software (proprietary and nonproprietary, excluding in-house developed applications), and network components. The best tools and migration methods are determined, and a project plan that maps the steps from the current to the future state is created.

1.5.4 Application Migration Service

The Application Migration service migrates an in-house developed application from a VAX platform to an Integrity platform. Each code module is either recompiled or translated, depending on source code availability. First, VAX dependencies are removed. Then, the entire application is relinked and tested on the Integrity platform. The application is then deployed on the target systems.

1.5.5 System Migration Service

The System Migration service migrates an OpenVMS system (single node or cluster) from the VAX platform to the Integrity platform. The customer's system availability and performance requirements are reviewed, and acceptance testing methodology and criteria are determined.

2
Selecting a Migration Method

Evaluating your application identifies the work to be done and allows you to plan the rest of the migration.

The evaluation process has three main stages:

- General inventory, including identifying dependencies on other software
- Source analysis to identify coding practices that affect migration
- Selection of a migration method: rebuilding from source code or translating

When you have completed these steps, you will have the information necessary to write an effective migration plan.

2.1 Taking Inventory

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
  - CLD files
  - UIL and UID files for DECwindows support
- Other software on which your application depends, for example:
  - Run-time libraries
  - HP layered products
  - Third-party products

  To help identify dependencies on other code, use VEST with the qualifier /DEPENDENCY. The VEST/DEPENDENCY command identifies executable and shareable images on which your application depends, such as run-time libraries, system services, and other applications. For information about using VEST/DEPENDENCY, see *OpenVMS Migration Software for VAX to Alpha Systems: Translating Images*.

- 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

    Includes HP tools such as Code Management System (CMS) and Module Management System (MMS).
  - Testing suite

    Your tests help confirm that the migrated application runs correctly and help evaluate its performance.

Many items, such as the following, have already been migrated to OpenVMS I64:

- HP software bundled with OpenVMS
  - RTLs
  - Other shareable libraries, such as those supplying callable utility routines and application library routines
- HP layered products
  - Compilers and compiler RTLs
  - Database managers
  - Networking environment
- Third-party products

  Many third-party applications now run on OpenVMS 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 testing suites.

2.2 How to Select a Migration Method

After you complete the inventory for your application, you must decide how to migrate each part of it: by recompiling and relinking or by translating. The majority of applications can be migrated by recompiling and relinking them. If your application runs only in user mode and is written in a standard high-level language, it is probably in this category. For the major exceptions, see Section 2.5.

The remainder of this chapter discusses how to choose a migration method for the relatively few applications that require more work to migrate. 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.

Note

The following process assumes that you will recompile your application if possible, and that you will use translation only for parts that cannot be recompiled or as a temporary measure in the course of the migration.

Follow these steps to choose a migration method for your application:

1. Determine which of the two migration methods is possible.

   Under most conditions, you can either recompile and relink your program or translate the VAX image. Section 2.3 describes cases where only one migration method is available.

2. Identify architectural dependencies that affect recompilation.

   Even if your application is generally suitable for recompiling, it might contain code that depends on features of the VAX architecture that are incompatible with the Intel Itanium architecture.

   Section 2.5 discusses these dependencies and provides information that helps you to identify them and to begin estimating the type and amount of work required to accommodate any dependencies you find.

   Section 2.7 describes tools and methods you can use to help answer the questions that come up in evaluating your application.

3. Decide whether to recompile or translate.

   After you have evaluated your application, you must decide which migration method to use. Section 2.4 describes how to make the decision by balancing the advantages and disadvantages of each method.

   If you cannot recompile and relink your program, or if the VAX image uses features specific to the VAX architecture, you may wish to translate that image. Section 2.4.1 describes ways to increase the compatibility and performance of translated images.

As shown in Figure 2–1, the evaluation process consists of a series of questions and some tasks you can perform to help answer those questions. HP provides a number of tools that you can use to help answer the questions; these tools are described at the relevant points in the process.

2.3 Which Migration Methods are Possible?

In most cases, you can either recompile and relink, or translate your application. However, depending on the design of your application, only one of the two migration paths may be available to you:

- Programs that cannot be recompiled

  The following types of images must be translated:

  – Software that is written in a programming language for which no I64 compiler is yet available

  – Executable and shareable images for which the source code is not available

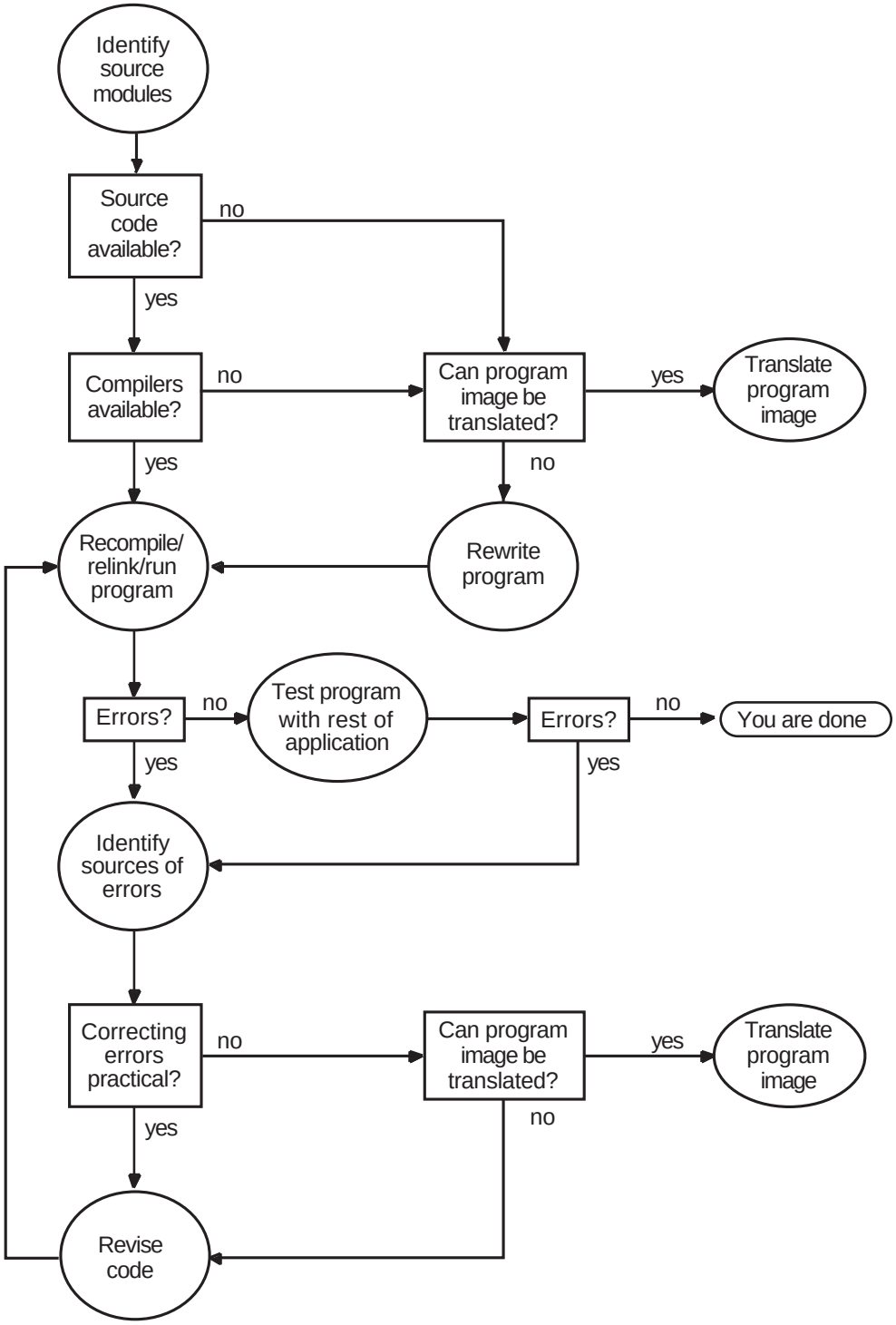
  – Programs that require H_floating or 56-bit D_floating data

- Images that cannot be translated

  The source code must be recompiled and relinked (and possibly revised) for the following types of images:

  – Images produced before OpenVMS VAX Version 4.0

Figure 2–1 Migrating a Program



- Images produced using OpenVMS VAX Version 5.5 or later, because the translated RTLs and system services have not been updated since then
- Images written in Ada
- Images that call or are called by images written in Ada
- Images that use PDP-11 compatibility mode
- Based images
- Images that contain coding practices intended for the VAX architecture. These images include code that:
  - Runs in inner access modes or elevated IPL (for example, VAX device drivers)
  - Refers directly to addresses in system space
  - Refers directly to undocumented system services
  - Uses threaded code; for example, code that switches stacks
  - Uses VAX vector instructions
  - Uses privileged VAX instructions
  - Inspects or modifies return addresses or makes other decisions based on a program counter (PC)
  - Depends on exact access-violation behavior due to 512-byte size memory page dependencies
  - Aligns global sections on boundaries other than the native machine page boundary (for example, depends on a 512-byte memory page size)
  - Uses most of the VAX P0 or P1 space or is otherwise sensitive to the space taken up by the translated-image run-time support routines

Although the translated image's run-time performance will be degraded because of the amount of VAX code that TIE is required to interpret, VEST can probably translate the following kinds of images:

- Images that include self-modifying or created VAX code, except for the code generated at run time by TIE
- Images with code that inspects the instruction stream, except when TIE interprets such code at run time

For more information about which images can be translated, see *OpenVMS Migration Software for VAX to Alpha Systems: Translating Images*.

2.4 Deciding Whether to Recompile or Translate

If both methods are possible for a given image, you must balance the projected performance of native and translated versions of the image on an I64 system against the effort required to translate the image or to convert it to a native I64 image.

In general, different images that make up an application can be run in different modes. For example, a native I64 image can call translated shareable images and vice versa. For more information about mixed-architecture applications, see Section 2.4.2.

Table 2–1 compares the two migration paths.

Table 2–1 Migration Path Comparison

Factor	Recompile/Relink	Translate
Performance	Full I64 capability	Typically 25-40% of native I64 potential; equivalent to performance on VAX.
Effort required	Varies: easy to difficult.	Easy.
Schedule constraints	Based on availability of native compilers.	None: available immediately.
Programs supported:		
–Age	Source for VAX/VMS Version 4.0 or earlier accepted.	Only VAX/VMS Version 4.0 or later supported.
–Limitations	Privileged code supported.	Only user-mode code supported.
VAX compatibility	High: most code recompiles and relinks without difficulty.	Complete by emulation.
Ongoing support and maintenance.	Normal source code maintenance	Maintain source code on VAX; recompile and retranslate for each new version.

To determine how to proceed with the migration of your application, answer the following questions:

- Do you build your application entirely from source code, or do you rely on binary images for some functions?

  If you rely on binary images, you must translate them.

- Do you have access to the source code for all images that are part of your application?

  If not, you must translate images that are missing source code.

- Which images are critical to the performance of your application?

  You should recompile these images to take full advantage of the speed of I64 systems.

  – Use the PCA to identify critical images.

  – Only images that are produced by native I64 compilers use I64 processing capabilities efficiently and achieve optimal performance. A translated VAX image runs at one-third the speed of native I64 code or slower, depending on the translation options used.

- How much work is required to convert each image under the two methods?

  – Translating VAX images to I64 images is a two-step process:

    1. Translate the VAX images to Alpha images using the VEST utility.
    2. Translate the Alpha images to I64 images using the HP OpenVMS Migration for Alpha to Integrity Servers (OSMAI) utility.

– Code that depends on details of the VAX architecture and the VAX calling standard cannot be recompiled directly. It must either run under translation, or it must be rewritten, recompiled, and relinked.

You can remove architectural dependencies in several ways:

- Replace an architecture-dependent code sequence with high-level language lexical elements that support the same operation in a platform-independent manner.
- Use a call to an OpenVMS system service to perform the task in a way that is appropriate for the processor architecture.
- Use a high-level language compiler switch to help guarantee correct program behavior with minimal changes to the source code.

Table 2–2 summarizes how the architectural dependencies of a given program can affect which method you use to migrate the program to an I64 system. For more detailed information, see the following chapters.

Table 2–2 Choice of Migration Method: Dealing with Architectural Dependencies

Recompiled, Relinked VAX Source	Translated VAX Image
Data alignment[1]	
By default, most compilers align data naturally. For information about qualifiers to retain VAX alignment, see Chapter 9.	Unaligned data supported, but the qualifier /OPTIMIZE=ALIGNMENT can improve overall execution speed by assuming that data is longword aligned.
Data types	
Replace H_floating with X_floating. For D_floating, if the 15 decimal digits of precision provided by the D53 format are sufficient, replace D_floating with G_floating. If the application requires 16-bit decimal precision (D56 format), translate it. COBOL packed decimal is automatically converted to binary format for operations. For more information about data types, see Chapter 7.	To retain 16-bit decimal precision for D_floating, use the /FLOAT=D56_FLOAT qualifier. Performance using this qualifier will be slower than when using the default, /FLOAT=D53_FLOAT.
Atomicity of read-modify-write operations	
Support depends on options provided by the individual compiler. (for more information, see Chapter 6)	Use the /PRESERVE=INSTRUCTION_ ATOMICITY qualifier. Execution speed may drop by a factor of 2.

[1]Unaligned data is primarily a performance issue. Whereas references to unaligned data are only somewhat detrimental to VAX performance, loading unaligned data from memory and storing unaligned data to memory in an I64 system can be up to 1000 times slower than the corresponding aligned operations.

(continued on next page)

Table 2–2 (Cont.) Choice of Migration Method: Dealing with Architectural Dependencies

Recompiled, Relinked VAX Source	Translated VAX Image
Page size	
The OpenVMS Linker produces large, I64-style pages by default.	Most 512-byte page images are supported. However, because of the permissive protection assigned by VEST, images that rely on restrictive protection to generate access violations cannot execute properly on an I64 system when translated.
Read/write ordering	
Supported by adding appropriate synchronization instructions (MF) to source code, but with a performance penalty. (for more information, see Chapter 6)	Use the /PRESERVE=READ_WRITE_ ORDERING qualifier.
Explicit reliance on details of the VAX architecture and calling standard[2]	
Unsupported; dependencies must be removed.	Supported.

[2]Dependencies on details of the VAX architecture and calling standard include explicit reliance on the VAX calling standard, VAX exception handling, the VAX AST parameter list, the format and behavior of VAX instructions, and the generation of VAX instructions at run time.

2.4.1 Translating Your Application

If you are unable to recompile your application, or if it uses features specific to the VAX architecture, you might decide to translate the application. You can translate only some parts of the application, or you can translate parts of it temporarily as a means of staging the overall migration.

Many of the differences that affect recompilation discussed in Section 2.5 can also affect the performance of a translated VAX image. For more information, see *OpenVMS Migration Software for VAX to Alpha Systems: Translating Images* and *HP OpenVMS Migration Software for Alpha to Integrity Servers: Guide to Translating Images*.

Table 2–2 includes a summary of ways you can allow for various architectural dependencies in a translated image.

2.4.2 Combining Native and Translated Images

In general, you can combine native I64 images with translated images on an I64 system. For example, a native I64 image can call translated shareable images, and vice versa.

In order to run together, calls between native and translated images must be able to account for the calling standard differences between the VAX and Integrity platforms.

- Routine interface semantics and data alignment conventions for the native I64 images are identical to those for VAX images.

- All the entry points are CALL*x*; there are no external JSB entry points. This is probably true of any code written in a high-level language.
- The inbound and outbound calls in the native image are not written in Ada.

When a translated image makes a call to a routine in a native image, or vice-versa, it does so indirectly through a **jacket routine**. The jacket routine interprets the procedure's call frame and argument list and builds the equivalent destination call frame and argument list, then transfers control to the destination procedure. When the destination procedure returns, it does so through the jacket routine. The jacket routine propagates the destination routine's returned register values into the source routine's registers and returns control to the source procedure.

The OpenVMS I64 operating system creates jacket routines automatically for most calls. To make use of automatic jacketing, use the compiler qualifier /TIE and the linker option /NONATIVE_ONLY to create the native I64 parts of your application.

In certain cases, the application program must use a specially written jacket routine. For example, you may have to write jacket routines for nonstandard calls to libraries such as the following:

- A VAX shareable library that includes external JSB entry points
- A library that includes read/write data in the transfer vector
- A library that contains VAX specific functions
- A library that uses resources that would need to be shared between a native and a translated version of the library
- A native I64 library that does not provide or export all the symbols that the VAX image did

  (The term exported means that a routine is included in the Global Symbol Table [GST] for the image.)

For information about how to create a jacket image for one of these situations, see *OpenVMS Migration Software for VAX to Alpha Systems: Translating Images*.

Translated shareable images (such as run-time libraries for languages without native I64 compilers) that are shipped with the OpenVMS I64 operating system are accompanied by jacket routines that allow them to be called by native I64 images.

2.5 Coding Practices That Affect Recompilation

Many applications, especially those that use only standard coding practices or are written with portability in mind, can migrate from OpenVMS VAX to OpenVMS I64 with little or no trouble. However, recompiling an application that depends on VAX-specific features that are incompatible with the Intel Itanium architecture require you to modify your source code. Typical incompatibilities include use of the following:

- VAX MACRO assembly language to obtain high performance on a VAX system or to make use of features specific to the VAX architecture
- Privileged code
- Features specific to the VAX architecture

If none of these incompatibilities is present in your application, the rest of this chapter does not apply to you.

2.5.1 VAX MACRO Assembly Language

On I64 systems, VAX MACRO is not the assembly language, but just another compiled language. However, unlike the high-level language I64 compilers, the VAX MACRO–32 Compiler for OpenVMS I64 does not produce highly optimized code in all cases. HP strongly recommends that you use the VAX MACRO–32 Compiler for OpenVMS I64 only as a migration aid, not for writing new code.

Many of the reasons for using assembly language on a VAX system are no longer relevant on I64 systems, for example:

- There is no inherent performance advantage in using assembly language on a EPIC processor. EPIC compilers, such as those in the I64 compiler set, can generate optimized code that takes advantage of architecture- and implementation-specific features more easily and efficiently than a programmer can.
- New system services can perform some functions that previously required assembly language.

For more information about migrating MACRO code, see *HP OpenVMS MACRO Compiler Porting and User's Guide*.

2.5.2 Privileged Code

VAX code that executes in inner access mode (kernel, executive, or supervisor mode) or that references system space is more likely to use coding practices that is VAX architecture specific or that refer to VAX data cells that do not exist on OpenVMS I64. Such code will not migrate to an I64 system without change. These programs require recoding, recompiling, and relinking.

Code in this category includes:

- User-written system services and other privileged shareable images

  For more information, see the *HP OpenVMS Programming Concepts Manual* and the *HP OpenVMS Linker Utility Manual*.
- Device drivers and performance monitors not supplied by HP
- Code that uses special privileges; for example, code that uses $CMEXEC or $CMKRNL system services, or code that uses the $CRMPSC system service with the PFNMAP option

  For more information about memory mapping, see Chapter 5.
- Code that uses internal OpenVMS routines or data, such as:
  - Code that links against the system symbol table, SYS.STB, to access locations in system address space
  - Code that compiles against SYS$LIBRARY:LIB

For assistance in migrating inner-mode code that refers to the OpenVMS executive, contact an HP support representative.

2.5.3 Features Specific to the VAX Architecture

To achieve its high performance, the Intel Itanium architecture differs significantly from the VAX architecture. Software developers who are accustomed to writing code that relies on certain aspects of the VAX architecture must be aware of architectural dependencies in their code in order to transport it successfully to an I64 system.

Common architectural dependencies, along with ways to identify them and actions you can take to eliminate them, are described briefly in the following sections. For a detailed discussion of ways to identify and eliminate these dependencies, see Chapters 4 through 8.

2.6 Identifying Dependencies on the VAX Architecture in Your Application

Even if your application recompiles successfully with a compiler that generates native I64 code, it might still contain subtle dependencies on the VAX architecture. The OpenVMS I64 operating system has been designed to provide a high degree of compatibility with OpenVMS VAX; however, the fundamental differences between the VAX and Intel Itanium architectures can create problems for applications that depend on certain VAX architectural features. The following sections highlight areas of your application you should examine.

2.6.1 Data Alignment

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.

Accessing data that is not naturally aligned in memory incurs a significant performance penalty on both VAX and I64 systems. On VAX systems, most languages align data on the next available byte boundary by default, because the VAX architecture provides hardware support that minimizes the performance penalty in referencing unaligned data. On I64 systems, however, the default is to align each data item naturally for better performance. As a result, references to naturally aligned data on Intel Itanium systems are 10 to 1000 times faster than references to unaligned data.

I64 compilers automatically correct most potential alignment problems and flag others.

Finding the Problem

To find instances of unaligned data, you can use the following methods:

- Use a qualifier provided by most I64 compilers that allows the compiler to report compile-time references to unaligned data. For example, for HP Fortran programs, compile with the /WARNING=ALIGNMENT qualifier.
- To detect unaligned data at run time, use the OpenVMS Debugger (SET BREAK/UNALIGNED command) or DEC PCA (Performance and Coverage Analyzer).

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. For example, in HP C, natural alignment is the default option. To define data structures that must match the VAX C default alignment—such as on-disk data structures—use the construct #PRAGMA NO_MEMBER_ALIGNMENT. With HP Fortran, local variables are naturally aligned by default. To control alignment of record structures and common blocks, use the /ALIGN qualifier.
- Use compiler qualifiers that generate VAX compatible unaligned data-structure mappings. Use of these qualifiers results in I64 programs that are functionally correct but potentially slow.

Note

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

- An existing file format might specify records with unaligned data.
- A translated image might 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.

For more information about data alignment, see Chapter 7 and Section 8.4.2.

2.6.2 Floating-Point Arithmetic

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

The VAX architecture supports VAX floating-point formats in hardware. The Intel Itanium architecture implements floating-point ,arithmetic in hardware using the IEEE floating-point formats, including IEEE single and IEEE double.

If an application was originally written for OpenVMS VAX or OpenVMS Alpha using the default floating-point formats, it can be ported to OpenVMS I64 in one of two ways: continue to use VAX floating-point formats utilizing the conversion features of the HP compilers, or convert the application to use IEEE floating-point formats. VAX floating-point formats can be used in situations where access to previously generated binary floating-point data is required. HP 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. See the Related Documents section in the Preface of this manual for the Web location of this white paper.

IEEE floating-point format should be used in situations where VAX floating-point formats are not required. The use of IEEE floating-point format results in more efficient code.

2.6.3 Data Types

The Intel Itanium architecture supports many of the VAX native data types; however, certain VAX data types, such as the H_floating data type, are not supported at all, while other data types such as F-floating, are supported by converting to IEEE format to perform the desired operation (see Table 2–3). Check to see if your application depends on the size or bit representation of an underlying native data type.

Table 2–3 Floating-Point Data Type Support

Data Type	On VAX	On I64
G_floating	Supported.	Supported by converting to IEEE T-floating automatically if /FLOAT=G_FLOAT is specified on compile command. Using D53_floating instead of D56_floating drops 3 bits of precision and yields slightly different results.
D_floating	Supported.	Supported by converting to IEEE T-floating automatically if /FLOAT=D_FLOAT is specified on compile command.
F_floating	Supported.	Supported by converting to IEEE S-floating automatically if /FLOAT=D_FLOAT or /FLOAT=G_FLOAT is specified on compile command.
H_floating (128-bit floating-point)	Supported.	Not supported. You can obtain full H_floating support with DECmigrate. You can use it to translate the code module that contains H_floating structures, or you can recode your application, using a supported data type.
S_floating (IEEE)	Not supported.	Supported.
T_floating (IEEE)	Not supported.	Supported.
X_floating (128-bit floating-point (Itanium; IEEE-like))	Not supported.	Supported. The X_floating data format is not identical to H_floating, but both cover a similar range of values. For Fortran applications, automatic conversion between X_floating memory format and H_floating on-disk is possible by use of the FOR$CONVERT*nnn* logical name, the OPTIONS statement, the /CONVERT compiler qualifier, or the CONVERT=*keyword* on OPEN statements.

To improve their performance, Intel Itanium processors implement the numeric string and packed decimal string, H_floating, G_floating, D_floating and F_ floating data types by using software, as follows:

- Decimal

  Eighteen-digit decimal data is converted to 64-bit binary integers internally, which provides very fast COBOL performance.

- H_floating

  I64 compilers do not support H_floating data; however, the Translated Image Environment (TIE) provides emulated support for H_floating data in translated images.

- D_floating, G_floating, F_floating

  Each VAX floating data type is converted to an equivalent IEEE value before performing the requested operation. Because of slight differences in range and accuracy, the exact answer might differ slightly from VAX.

Eliminating the Problem

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

- Instead of D_floating, F_floating, G_floating or H_floating, use IEEE S_floating or IEEE T_floating whenever possible.
- Instead of decimal data types, use integer data types whenever possible.

For more information about I64 data types, see Chapter 7. Specifically, Section 7.2 discusses how to check for dependence on a VAX data type.

For more information about floating-point data types, see the white paper "OpenVMS Floating-Point Arithmetic on the Intel® Itanium® Architecture". The Preface of this manual provides the Web location of this white paper.

2.6.4 Shared Access to Data

An **atomic operation** is one in which:

- Intermediate or partial results cannot be seen by other processors or devices.
- The operation cannot be interrupted (that is, once started, the operation continues until completion).

On OpenVMS I64, any operation that reads, modifies, and stores data in memory is broken into several instructions and can be interrupted between any of those instructions. As a result, if your application modifies data in shared memory atomically, you must take steps to guarantee the atomicity of the operation.

An application can depend on the atomicity of operations under any of the following conditions:

- An AST routine within the process shares data with the mainline code.
- The process shares data in a **writable global section** with another process that executes on the same CPU (that is, in a uniprocessor system).
- The process shares data in a writable global section with another process that might execute concurrently on another CPU (that is, in a multiprocessor system).

Finding the Problem

To find dependencies on atomicity, reexamine use of shared variables (writable items accessed by multiple threads of execution) for hidden or explicit assumptions of atomicity.

Eliminating the Problem

To eliminate general problems of instruction atomicity, you can use one or more of the following methods:

- Use language constructs, where available, that guarantee atomicity to protect shared variables, for example, in C, use the VOLATILE declaration.
- Use explicit **synchronization** rather than relying on assumptions of atomicity.
- Use OpenVMS locking services (such as $ENQ and $DEQ) or LIB$ routines.

- To synchronize with an AST thread, use the $SETAST system service in the mainline code to block the AST, and then reenable delivery after the instruction has completed.

For more information about synchronization, see Chapter 6.

2.6.5 Page Size Considerations

Page size governs the amount of virtual memory allocated by memory management routines and system services. For example, in mixed-architecture OpenVMS Cluster systems, your application might determine the size of certain data buffers based on the VAX page size. Page size is also the basis on which protection is assigned to code and data in memory.

The OpenVMS VAX operating system allocates memory in multiples of 512 bytes. The page sizes on I64 depend upon SYSGEN parameter settings, not hardware platform characteristics. Note that this is a run-time setting and that the value can change from one boot to the next.

Page size is a factor in the management of system resources, such as working set quotas. In addition, memory protection on VAX systems can vary for each 512-byte region of memory; on I64 systems, the granularity of memory protection is much larger, depending on the system's page-size implementation.

Note

The change to a larger page size affects only applications that explicitly rely on a 512-byte page size. Examples of such applications are those that:

- Use "512" to:
  - Compute memory usage.
  - Calculate page table requirements.
- Change protection on a 512-byte granularity.
- Use the system service Create and Map Section ($CRMPSC) to map a file into a specific location in the process address space (for example, to reuse memory when available memory is limited).

Finding the Problem

To find uses of the VAX page size, identify code that manipulates virtual memory in 512-byte chunks or relies on 512-byte memory protection granularity. Search your code for occurrences of numbers such as the decimal values 511, 512, or 513; the hexadecimal values 1FF, 200, or 201; and so forth.

Eliminating the Problem

To eliminate conflicts between the VAX and I64 page sizes, you can use one or more of the following methods:

- Change hardcoded page size references to symbolic values (assigned at run time using a call to $GETSYI).
- Reevaluate code that assumes that page size and disk (file) block size are equal. On I64 systems, this assumption is not correct.

- Do not depend on being able to use memory-management-related system services (for example, $CRMPSC, $MGBLSC) to map a file into a fixed, page-size-dependent range of addresses (global section). Instead consider using the SEC$M_EXPREG flag.

For more information about page size, see Chapter 5.

2.6.6 Order of Read/Write Operations on Multiprocessor Systems

The VAX architecture specifies that if a processor in a multiprocessing system writes multiple data items to memory, these items are written to memory in the order specified. This ordering ensures that the writes become visible to other CPUs in the order in which they were specified by the program and I/O devices.

This guarantee limits the performance optimization that the system can make. It also makes caches more complex and limits the optimization of cache performance.

To benefit overall system performance, Integrity server systems, as well as other EPIC systems, can reorder reads and writes to memory. Therefore, writes to memory can become visible to other CPUs in the system in an order different from that in which they were issued.

Note

This section is relevant only to multiprocessor systems. On a uniprocessor system, all memory accesses are completed in the order in which the program requested them.

Finding the Problem

To find instances of reliance on read/write ordering for applications that may execute on multiprocessor systems, identify algorithms that depend upon the order in which data is written; for example, use of flag-passing protocols for synchronization.

Eliminating the Problem

To eliminate problems with the ordering of read and write operations, you can use one or more of the following methods:

- Instead of flag-passing protocols, use system-supplied routines for synchronization, such as the OpenVMS locking system services ($ENQ, $DEQ).
- The Intel Itanium architecture specifies a memory fence instruction, which causes the hardware to complete all previous memory reads and writes before performing reads and writes following the barrier. Some I64 languages provide a way of inserting this instruction, but its use degrades performance.

For more information about synchronization, see Chapter 6.

2.6.7 Explicit Reliance on the VAX Procedure Calling Standard

The OpenVMS Calling Standard specifies significantly different calling conventions for I64 programs than for VAX programs. Application programs that depend explicitly on certain details of the VAX procedure calling conventions must be modified to run as native code on an I64 system. Such dependencies include:

- Code that locates the placement of arguments relative to the argument pointer (AP)

  In many cases, however, the VAX MACRO–32 Compiler for OpenVMS I64 compensates for this.

- Code that modifies its return address on the stack
- Code that interprets the contents of a call frame

Both VAX and I64 compilers provide techniques for accessing procedure arguments. If your code uses these standard mechanisms, you can simply recompile it to run correctly on an I64 system. If your code does not use these mechanisms, you must rewrite it so that it does. For a description of these standard mechanisms, see the *HP OpenVMS Calling Standard*.

Translated code mimics the behavior of VAX procedure calling. Images that contain the dependencies listed here execute properly under translation on an I64 system.

2.6.8 Explicit Reliance on VAX Exception-Handling Mechanisms

The mechanics of exception handling differ between VAX and Integrity server systems. Chapter 8 discusses the differences in how arithmetic exceptions are dispatched on VAX and Integrity systems. This section focuses on the mechanisms by which code dynamically establishes a condition handler and by which a condition handler accesses the exception state.

2.6.8.1 Establishing a Dynamic Condition Handler

VAX systems provide a number of ways in which an application can establish a condition handler dynamically at run time. The OpenVMS Calling Standard facilitates this operation for VAX programs by providing a space at the top of a call frame in which executing code can place the address of a condition handler that is to service exceptions that occur in the context of that frame. However, the OpenVMS Calling Standard provides no such writable area for I64 procedures.

For instance, a VAX MACRO program might use the following instruction sequence to move the address of a condition handler into a call frame:

```
MOVAB    HANDLER,(FP)
```

The MACRO–32 Compiler for OpenVMS I64 parses this statement and generates appropriate I64 assembly language code that results in the establishment of the condition handler. For more information, see the *HP OpenVMS MACRO Compiler Porting and User's Guide*.

Note

On VAX systems, the run-time library routine LIB$ESTABLISH and its counterpart LIB$REVERT allow an application to establish and disestablish a condition handler for the current frame. These routines do not exist on I64 systems; however, compilers might handle these calls properly (such as with Fortran intrinsic functions). For more precise information, see Chapter 9 and the documentation for the compilers relevant to your application.

Translated code mimics the VAX mechanism for dynamically establishing a condition handler.

Certain I64 compilers provide a language-specific mechanism to establish a dynamic condition handler.

For more information about condition handlers, see Chapter 8.

2.6.8.2 Accessing Data in the Signal and Mechanism Arrays

During exception processing, both VAX and I64 systems push the exception processor status, an exception PC, a signal array, and a mechanism array onto the stack.

Both the signal array and mechanism array have different contents on VAX and I64 systems; the mechanism array also has different formats on the two platforms. To work properly in either system, a condition handler that accesses data in either the signal array or the mechanism array must use the appropriate CHF$ symbols rather than hardcoded offsets. For descriptions of the appropriate CHF$ symbols, see the *HP OpenVMS Programming Concepts Manual*.

Note

The condition handler cannot successfully locate information in the mechanism array by using hardcoded offsets from AP.

2.6.9 Modification of the VAX AST Parameter List

OpenVMS VAX passes five longword arguments to an AST service routine. AST service routines written in VAX MACRO access this information by using offsets from the argument pointer (AP). OpenVMS VAX allows an AST service routine to modify any of these arguments, including the saved registers and the return PC. These modifications can then affect the interrupted program once the AST routine returns.

Although the AST parameter list on I64 systems also consists of five parameters, the only argument directly intended for the AST procedure is the AST parameter. Although the other arguments are present, they are not used after the AST procedure exits. Because modifying them has no effect on the thread of operation to be resumed at AST exit, a program that relies on such an effect must be changed to use more conventional argument-passing mechanisms to run on an I64 system.

2.6.10 Explicit Dependency on the Form and Behavior of VAX Instructions

Programs that rely specifically on the execution behavior of VAX MACRO instructions or on binary encoding of VAX instructions must be modified before being recompiled or relinked to run as native code on an I64 system.

For example, the following coding practices do not work on I64 systems:

- In VAX MACRO, embedding a block of VAX instructions in a program data area, and modifying a PC to transfer control to this code block
- Examining condition codes or other information in the processor status longword (PSL)

For more information about migrating VAX MACRO code, see the *HP OpenVMS MACRO Compiler Porting and User's Guide*.

2.6.11 Generation of VAX Instructions at Run Time

Creating and executing conventional VAX instructions do not work in native I64 mode. VAX instructions created at run time execute by emulation in a translated image.

For more information about code that generates specific VAX instructions at run time, see the *HP OpenVMS MACRO Compiler Porting and User's Guide*.

2.7 Identifying Incompatibilities Between VAX and I64 Systems

To identify architectural incompatibilities in a module of your application, start by doing a test compile of the module using the I64 compiler. For information about diagnostic compiler switches, see your language processor documentation.

Many modules compile and run with no errors. If errors occur, you might have to revise the module.

The HP compilers can produce messages that are useful for identifying porting problems. For example, the MACRO–32 compiler provides the /FLAG qualifier with several options. Depending on which options you include, the compiler reports all unaligned stack and memory references, any run-time code generation (such as self-modifying code), branches between routines, and several other conditions.

The HP Fortran compiler qualifier, /CHECK, produces messages about any of the various options you specify.

Note

Even if a module runs without error in isolation, latent synchronization problems might be present that will be detected only when the module is run together with other parts of the application.

If a module does not run without error after being recompiled and relinked, you can use the following methods to assess what must be revised to make the program run well on an I64 system:

- Examining the source code

  A code review at this point can avoid many difficulties in the migration process and save a great deal of time and effort in the later stages of migration. To examine your code, use the checklist in Appendix A, and the guidelines in Chapter 4. These migration issues are summarized in Section 2.5.

  If a direct code review of your entire application is not practical, an automated search can still be useful, for example, you can use a combination of the DCL SEARCH command and an editor to locate and tabulate instances of architectural dependencies.

- Using messages generated by the compiler in your initial test run

  Compilers give you information about the following:

  – Differences between VAX and I64 compilers

  – Data alignment

  Specific compilers might also identify other differences between the VAX and Intel Itanium architectures.

- Analyzing the image using VEST

  Even if you intend to recompile and relink a program, you can use VEST as an analysis tool. It can provide a great deal of useful information about changes that can make your program run most efficiently on an I64 system. For example, VEST can help identify the following problems:

  - Static unaligned data (data declarations, including unaligned fields in data structures) and unaligned stack operations
  - Floating-point references (H_floating and D_floating)
  - Packed decimal references
  - Privileged code
  - Nonstandard coding practice
    - References to OpenVMS data or code other than by using system services
    - Uninitialized variables
  - Certain synchronization issues, such as multiprocess use of interlocked instructions

  VEST cannot identify some problems, including:

  - Unaligned variables (in data structures created dynamically)
  - Most synchronization issues

- Running the image using the PCA (Performance and Coverage Analyzer)

  The PCA can point out the following issues:

  - Run-time alignment faults
  - Which sections of the application are executed most frequently and, hence, are critical to performance

Once all the images of the application run without errors on the I64 system, you must combine the rebuilt images to test for problems of synchronization between images. For more information about testing, see Section 3.5.

3
Migrating Your Application

Actually migrating your application to an OpenVMS I64 system involves several steps:

1. Setting up the migration environment
2. Testing the application on a VAX system to establish baselines for evaluating the migration
3. Converting the application to run on an I64 system
4. Debugging and testing the migrated application
5. Integrating the migrated application into a software system
6. Modifying certain types of code

3.1 Setting Up the Migration Environment

The native I64 environment is a complete development environment equivalent to that on VAX systems. You must compile, link, debug and test a migrated application on the Integrity server system.

An important element of the I64 migration environment is support from HP, which can provide help in modifying, debugging, and testing your application.

3.1.1 Hardware

You should consider several issues when you plan what hardware you need for your migration. To begin, consider what resources are required in your normal VAX development environment:

- CPUs
- Disks
- Memory

To estimate the resources needed for an I64 migration environment, consider the following issues:

- Greater image size on I64 systems

  Compare VAX and I64 compiled and translated images.
- Greater page size and physical memory size on I64 systems
- CPU requirements

Translating a VAX image to run on an Integrity server tends to use a lot of CPU time. (It is difficult to predict how much; it depends more on application complexity than on size.) The image translators also need a great deal of disk space for log files, for an Alpha image if you request one, for flowgraphs, and so on. The new image includes the original VAX instructions, the translated Alpha

instructions, and the new Itanium instructions, so it is always larger than the VAX image.

A suggested configuration consists of:

- 6 VUP multiprocessing system with 256 MB of memory
- 1 GB system disk
- 1 GB disk per application

In a multiprocessing system, each processor should be able to support the image analysis of a separate application.

If computer resources are scarce, HP suggests that you do one or more of the following:

- Run compilers or VEST as a batch job at off-peak hours.
- Lease additional equipment for the migration effort.

3.1.2 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
    - VEST for translating a VAX image to Alpha
    - HP OpenVMS Migration Software for Alpha to Integrity Servers (OSMAIS)
  - RTLs
  - System libraries
  - Include files for C programs

- 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 OpenVMS VAX test tools to OpenVMS I64, unless they are already ported. You also need test tools that are designed to test your application for behaviors that are unique to the OpenVMS I64 platform. Test tools for Alpha might also be necessary, because if you translate first from VAX to Alpha and then to I64, you might need to verify that the first step was successful.

- Tools for maintaining sources and building images, such as:

  - CMS (Code Management System)
  - MMS (Module Management System)

Translation

The software translator OSMAIS runs on both Alpha and I64 systems. 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 either on an I64 system or at an I64 Migration Center.

3.2 Converting Your Application

If you have thoroughly analyzed your code and planned the migration process, this final stage should be straightforward. You might be able to recompile or translate many programs with no change. Programs that do not recompile or translate directly usually need only straightforward changes in order to run on an I64 system.

For more detailed information about the actual conversion of your code, see the following OpenVMS I64 migration documentation:

- *OpenVMS Migration Software for VAX to Alpha Systems: Translating Images*
- *HP OpenVMS MACRO Compiler Porting and User's Guide*
- *HP OpenVMS Migration Software for Alpha to Integrity Servers: Guide to Translating Images*

For descriptions of these books, see the Preface of this manual.

3.2.1 Recompiling and Relinking

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.

Over time, changes occur in both language standards and compiler quality. Depending upon the portability of your code, you might encounter compiler messages that indicate stricter adherence to standards or other quality improvements. The more portable your code is, the fewer messages you are likely to encounter.

Your compile and link commands require some changes, such as new compiler and linker switches. For example, to allow portability among different versions of OpenVMS I64, the linker default page size is 64 KB, which allows OpenVMS I64 images to run on systems with a default page size of up to 64 KB.

Several native compilers and other tools are available for software development and migration on I64 systems.

3.2.1.1 Native I64 Compilers

Recompiling and relinking an existing VAX source produces a native I64 image that executes within the I64 environment with all the performance advantages of the Intel Itanium architecture. For I64 code, HP is using a series of highly optimizing compilers.

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

- GNAT Pro Ada[1]
- BASIC

[1] GNAT Pro, available from Ada Core Technologies, is the recommended Ada 95 compiler for OpenVMS I64. See Chapter 9 for more information.

- BLISS (available on the Freeware CD)
- C
- C++
- COBOL
- Fortran
- MACRO–32
- Pascal

Most VAX user-mode programs that are written in any other language[1] can be run on an I64 system by translating them with VEST and OSMAI. Compilers for other languages might be available through third-party vendors.

In general, the I64 compilers provide command-line qualifiers and language semantics to allow code with dependencies on the VAX architecture to run on an I64 system with little modification. For a list of such dependencies, see Table 2–2.

Some compilers on OpenVMS I64 systems support new features not supported by their counterparts on OpenVMS VAX systems. To provide compatibility, some compilers support compatibility modes. For example, the HP C compiler for OpenVMS I64 systems supports a VAX C compatibility mode that is invoked by specifying the /STANDARD=VAXC qualifier.

In some cases, the compatibility is limited. For example, VAX C implements built-in functions that allow access to special VAX hardware features. Since the hardware architecture of VAX computers differs from that of Integrity servers, these built-ins are not available in HP C for OpenVMS I64 systems, even when the /STANDARD=VAXC qualifier is used.

The compilers can also compensate for some architectural dependencies that exist in your code. For example, the MACRO–32 compiler provides the /PRESERVE qualifier that can preserve granularity or atomicity, or both.

The HP C compiler provides a header file that defines typedefs for each data type. These typedefs map a generic datatype name, such as int64, to the machine-specific datatype, such as __int64. For example, if you must have a datatype that is 64 bits long, use the int64 typedef.

Review the documentation for your compiler to become familiar with all its features that support portability.

Chapter 9 describes in greater detail the process of using I64 compilers to migrate OpenVMS VAX programs to an OpenVMS I64 system.

3.2.1.2 VAX MACRO–32 Compiler for OpenVMS I64

The VAX MACRO–32 Compiler for OpenVMS I64 is used to convert existing VAX MACRO code into machine code that runs on OpenVMS I64 systems. It is included with OpenVMS I64 for that purpose.

While some VAX MACRO code can be compiled without any changes, most code modules require the addition of entry point directives. Many code modules require other changes as well.

[1] PL/I programs cannot be translated.

Note

The MACRO–32 compiler attempts to call LIB$ESTABLISH if it is contained in the source code.

If MACRO–32 programs establish dynamic handlers by storing a routine address at 0(FP), they work correctly when compiled on an OpenVMS I64 system. However, you cannot set the condition handler address from within a JSB (Jump to Subroutine) routine; you can do so only from within a CALL_ENTRY routine.

The compiler generates code that is optimized for OpenVMS I64 systems, but many features of the VAX MACRO language that provide the programmer with a high level of control make it more difficult to generate optimal code for OpenVMS I64 systems. For new program development for OpenVMS I64, HP recommends the use of mid-level and high-level languages. For more information on the MACRO–32 compiler, see *HP OpenVMS MACRO Compiler Porting and User's Guide*.

3.2.1.3 I64 Development Tools

Several tools in addition to the compilers are available to develop, debug, and deploy native I64 applications. Table 3–1 summarizes these tools.

Table 3–1 OpenVMS Development Tools

Tool	Description
OpenVMS Linker	The OpenVMS Linker accepts I64 object files to produce an I64 image. For more information about the OpenVMS Linker, see Chapter 4.
OpenVMS DEBUG	OpenVMS DEBUG on OpenVMS I64 is a symbolic, source-level debugger with several graphical interfaces. It is designed for debugging user-mode applications and has the same commands and interfaces as on OpenVMS VAX. For more information about OpenVMS DEBUG, see the *HP OpenVMS Debugger Manual*.
DELTA Debugger	The DELTA debugger on OpenVMS I64 is a nonsymbolic, instruction-level debugger for debugging process-based applications that run in an elevated mode (supervisor, executive, or kernel). For more information, see the *HP OpenVMS Delta/XDelta Debugger Manual*.
System Code Debugger (SCD)	SCD is a graphical, symbolic, source-level debugger for debugging operating system and device driver code. For more information, see the *HP OpenVMS System Analysis Tools Manual*.
XDelta Debugger	The XDelta debugger is a nonsymbolic debugger that is used for debugging operating system and device driver code. For more information on XDELTA on OpenVMS I64, see the *HP OpenVMS Delta/XDelta Debugger Manual*.
OpenVMS Librarian utility	The OpenVMS Librarian utility creates I64 libraries.

(continued on next page)

Table 3–1 (Cont.) OpenVMS Development Tools

Tool	Description
OpenVMS Message utility	The OpenVMS Message utility allows you to supplement the OpenVMS system messages with your own messages.
IAS (Itanium Assembler)	The IAS assembler for OpenVMS I64 systems is the native assembler for Intel Itanium processors. This assembler is not bundled with the operating system but is included on the Open Source CDs that are shipped with the OpenVMS I64 distribution.
ANALYZE/IMAGE	The Analyze/Image utility can analyze I64 images.
ANALYZE/OBJECT	The Analyze/Object utility can analyze I64 objects.
DECset	DECset, a comprehensive set of development tools, includes the Language Sensitive Editor (LSE), the Digital Test Manager (DTM), Code Management System (CMS), and Module Management System (MMS).
Command Definition utility	The Command Definition utility (CDU) enables application developers to create DCL commands.
System Dump Analyzer (SDA)	SDA has been extended to display information specific to OpenVMS 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.

3.2.2 Translating

The process of translating a VAX image to run on an I64 system is described in detail in *OpenVMS Migration Software for VAX to Alpha Systems: Translating Images* and *HP OpenVMS Migration Software for Alpha to Integrity Servers: Guide to Translating Images*.

3.3 Analyzing System Crashes

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

3.3.1 System Dump Analyzer

The System Dump Analyzer (SDA) utility on OpenVMS I64 systems is almost identical to the utility provided on OpenVMS VAX systems. Many commands, qualifiers, and displays are identical, and some additional commands and qualifiers are available, including several for accessing functions of the Crash Log Utility Extractor (CLUE) utility. Some displays are adapted to show information specific to OpenVMS I64 systems, such as processor registers and data structures.

While the SDA interface has changed only slightly, the contents of VAX and I64 dump files and the entire process of analyzing a system crash from a dump differ significantly between the two systems. The I64 execution paths leave more complex structures and patterns on the stack than VAX execution paths do.

To use SDA on a VAX computer, you must first familiarize yourself with the OpenVMS Calling Standard for VAX systems. Similarly, to use SDA on an I64 system, you must familiarize yourself with the OpenVMS Calling Standard for I64 systems before you can decipher the pattern of a crash on the stack.

The changes to SDA include the following:

- The displays of the SHOW CRASH and SHOW STACK commands contain additional information that simplify debugging of fatal system exception bugchecks.
- The SHOW EXEC command display includes additional information about executive images if they were loaded using image **slicing**. Slicing is a function performed by the executive image loader for executive images and by the OpenVMS Install utility for user-mode images. Slicing an executive image (or a user-mode image) greatly improves performance by reducing contention for the limited number of translation buffer entries.
- The MAP command, a new SDA command, enables you to map an address in memory to an image offset in a map file.
- A new symbol, FPCR, has been added to the symbol table. This symbol represents a floating-point register.

3.3.2 Crash Log Utility Extractor

The Crash Log Utility Extractor (CLUE) is a tool for recording a history of crash dumps and key parameters for each crash dump and for extracting and summarizing key information. Unlike crash dumps, which are overwritten with each system crash and are available only for the most recent crash, the crash history file (on OpenVMS VAX) and the summary crash history file with a separate listing file for each crash (on OpenVMS I64) are permanent records of system crashes.

Table 3–2 shows the implementation differences between OpenVMS VAX and OpenVMS I64.

Table 3–2 CLUE Differences Between OpenVMS VAX and OpenVMS I64

Attribute	OpenVMS VAX	OpenVMS I64
Access method	Invoked as a separate utility.	Accessed through SDA.
History file	A cumulative file that contains a one-line summary and detailed information from the crash dump file for each crash.	A cumulative file that contains only a one-line summary for each crash dump. The detailed information for each crash is put in a separate listing file.
Uses in addition to debugging crash dumps	None.	CLUE commands can be used interactively to examine a running system.

3.4 Testing Applications on VAX for Baseline Information

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

3.5 Testing the Migrated Application

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

The first step in testing is to establish baseline values for your application by running your test suite on the VAX application, as described in Section 3.5.

Once your application is running on an I64 system, you should apply two types of tests:

- Standard tests used for the VAX version of the application
- New tests to check specifically for problems resulting from the change in architecture

3.5.1 VAX Tests Ported to I64

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

3.5.2 New I64 Tests

Although your standard tests are extremely helpful in 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

3.5.3 Uncovering Latent Bugs

Despite your best efforts, you might encounter bugs that were in your program all along but that never caused a problem on an OpenVMS VAX system.

For example, failure to initialize some variable in your program might have been benign on a VAX system but can produce an arithmetic exception on an I64 system. The same can 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 "hidden" bugs, but you should test your programs after porting them and before making them available to other users.

3.6 Integrating the Migrated Application into a Software System

After you migrate your application by recompiling or translating it, check for problems that are caused by interactions with other software and that might have been introduced during the migration.

Sources of problems in interoperability can include the following:

- I64 and VAX systems within an OpenVMS Cluster environment must use separate system disks. You must make sure that your application refers to the appropriate system disk.
- In a mixed-architecture environment, be sure that your application refers to the correct version of image names.
  - Native VAX and native I64 versions of an image must have the same name.
  - Translated versions of VAX images must have the suffix "_TV_AV" added to their names. Translated versions of Alpha images must have the suffix "_AV" added to their names.
- Recompiled images expect naturally aligned data, while translated images have VAX aligned data, which might not be I64 aligned data.

3.7 Modifying Certain Types of Code

The following coding practices and types of code require changes to be run on I64 systems:

- Code that has been conditionalized to run on Alpha or VAX systems
- Code that uses OpenVMS system services that have dependencies on the VAX architecture.
- Code with other dependencies on the VAX 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

3.7.1 Conditionalized Code

This section describes how to conditionalize OpenVMS code for migration to I64. This code will be compiled for both VAX and I64, or for VAX, Alpha, and I64. The symbol ALPHA is new, although the symbol EVAX has not been eliminated. You do not need to replace EVAX with ALPHA but you can if you want. The architecture symbols available for MACRO and BLISS are VAX, EVAX, ALPHA, and I64.

3.7.1.1 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 VAX systems, VAX equals 1 while Alpha, EVAX and I64 are undefined. On I64 systems, I64 equals 1 and VAX, EVAX, and ALPHA are undefined.

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

For Alpha-specific code:

```
.IF DF VAX
        .
        .
        .
.ENDC
```

For I64-specific code:

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

3.7.1.2 BLISS Sources

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

Note

The constructs %BLISS(*xxx*), %TARGET(*xxx*), and %HOST(*xxx*) are not recommended. You do not need to replace them, but you can if you want.

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 VAX and I64 systems.

For VAX-specific code:

```
%if VAX %then
        .
        .
        .
%fi
```

For I64-specific code:

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

3.7.1.3 C Sources

For C source files, the symbols `__vax`, `__VAX`, `__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 VAX-specific code, use the following:

```
#ifdef __vax
        .
        .
        .
#endif
```

For I64-specific code, use the following:

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

3.7.1.4 Existing Conditionalized Code

You must examine existing conditionalized code 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 I64 code, then you would add the following case:

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

However, if the existing VAX/EVAX conditionals 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. The reason is because Alpha and 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
```

3.7.2 System Services with VAX Architecture Dependencies

Certain system services that work well in applications on OpenVMS VAX do not port successfully to I64. The following sections describe these system services and their replacement services.

3.7.2.1 SYS$GOTO_UNWIND

For OpenVMS VAX, 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 OpenVMS I64. For more information, refer to the *HP OpenVMS Calling Standard*.

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

3.7.2.2 SYS$LKWSET and SYS$LKWSET_64

The SYS$LKWSET and SYS$LKWSET_64 system services have been modified. For more information, see Section 3.7.9.

3.7.3 Code with Other Dependencies on the VAX Architecture

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

3.7.3.1 Initialized Overlaid Program Sections

Initialized overlaid program sections are handled differently on I64 systems. On OpenVMS VAX systems, different portions of an overlaid program section might be initialized by multiple modules. This practice is not allowed on OpenVMS I64 systems.

3.7.3.2 Condition Handler Use of SS$_HPARITH

On OpenVMS VAX, the SS$_HPARITH system services or "error code" is signaled for a number of arithmetic error conditions. On OpenVMS I64, SS$_HPARITH is never signaled for arithmetic error conditions; instead, the more specialized SS$_FLTINV and SS$_FLTDIV error codes are signaled on OpenVMS I64.

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

3.7.3.3 Mechanism Array Data Structure

The mechanism array data structure on OpenVMS I64 is very different from the one on OpenVMS VAX. The return status code RETVAL has been extended to represent the return status register on both Alpha and I64 platforms. For more information, refer to the *HP OpenVMS Calling Standard*.

3.7.3.4 Reliance on VAX Object File Format

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

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

`http://www.caldera.com/developers/gabi`

The object file format also conforms to the I64 specific extensions described in the *Intel® Itanium® Processor-specific Application Binary Interface (ABI)*, May 2001 edition (document number 245270-003). Extensions and restrictions that are 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:

`http://www.eagercon.com/dwarf/dwarf3std.htm`

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

3.7.4 Code That Uses Floating-Point Data Types

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

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

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

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

1. Converts VAX floating-point formats to either IEEE single or IEEE double floating-point formats, 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 VAX and OpenVMS I64 and how these differences might affect ported applications, see Chapter 9 and refer to the "OpenVMS Floating-Point Arithmetic on the Intel® Itanium® Architecture" white paper. See the Related Documents section in the Preface for the Web location of this 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 I64 at process startup.

3.7.4.1 LIB$WAIT Problem and Solution

The use of LIB$WAIT system service in code ported to OpenVMS I64 can cause unexpected results, as shown in the following C example:

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

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

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

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

A better coding method is to include the LIBWAITDEF call (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 the /FLOAT qualifier.

```
#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);
```

3.7.5 Incorrect Command Table Declaration

Incorrect declaration of a command table in code that is ported to OpenVMS I64 can cause unexpected results. For example, for an application, CDU 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 might fail with the following error message:

```
%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 the following:

```
EXTERNAL DFSCP_CLD
```

The command table is incorrectly declared in a FORTRAN module 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, and I64).

3.7.6 Code That Uses Threads

OpenVMS I64 supports all the thread interfaces that have been supported on OpenVMS since thread support was first introduced. Most OpenVMS VAX code that uses threads can be ported to OpenVMS I64 without change. This section describes the exceptions. The major porting issue for code that uses threads is the usage of stack space. I64 code uses much more stack space than does equivalent VAX code. Therefore, a threaded program that works on VAX might get stack overflow failures on I64.

The default stack size is larger on OpenVMS I64 to help alleviate overflow problems. If the application requests a specific stack size, then the change to the default is irrelevant, and the application source code might need to be changed to request more stack. HP 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 extreme cases, 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).

HP recommends that you familiarize yourself with the most recent improvements to thread support in OpenVMS, as documented in the *HP OpenVMS Version 8.2 Release Notes*. One change to the POSIX Threads C language header file PTHREAD_EXCEPTION.H caused problems when porting an application that relied on its former behavior.

The DCL command THREADCP is not supported on OpenVMS 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 VAX. For more information, refer to the *HP OpenVMS DCL Dictionary*. The THREADCP command is documented in the *Guide to the 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
```

3.7.6.1 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 is displayed:

```
cma_delay (
     ^

%CC-W-LONGEXTERN, The external identifier name exceeds 31
characters; truncated to "CMA_DELAY_NEEDS_VAX_FLOAT_____".
```

If an object file that triggered such a warning is linked, the linker displays an undefined-symbol message for this symbol. (If a linker-produced image is subsequently executed, the calls to these routines 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.

3.7.7 Code with Unaligned Data

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

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

Because natural alignment is not always possible, OpenVMS I64 systems provide help to manage the impact of unaligned data references. 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 occur between threads of a single process, between a process and ASTs, or between several processes in a global section.

Finding the Problem

To find instances of unaligned data, you can use a qualifier provided by most I64 compilers that allows the compiler to report compile-time references to unaligned data. Table 3–3 lists these qualifiers.

Table 3–3 Compiler Switches for Reporting Compile-Time Reference

Compiler	Qualifier
BLISS	/CHECK=ALIGNMENT
C	/WARN=ENABLE=ALIGNMENT
Fortran	/WARNING=ALIGNMENT
HP 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 might 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 might specify records with unaligned data.
- A translated image might 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.

3.7.8 Code That Relies on the OpenVMS VAX Calling Standard

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

- No frame pointer (FP)
- Multiple stacks
- Only four registers preserved across calls
- Changes to familiar register numbers

For more information, see the *HP OpenVMS Calling Standard*.

3.7.9 Privileged Code

This section describes categories of privileged code that require examination and that might require modification.

3.7.9.1 Use of SYS$LKWSET and SYS$LKWSET_64

If your application uses the SYS$LKWSET or SYS$LKWSET_64 system service 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 greater 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 systems, the code, data, and linkage data referenced by the program need to be locked. On 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 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 greater 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 I64 and not on 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 the LIB$LOCK_IMAGE and LIB$UNLOCK_IMAGE routines (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$_LKWSETFUL is returned. If you encounter this status, you can increase the user's working set quota. Otherwise, you can split the image into two parts, one that contains the user mode code and another shareable image that contains the kernel mode code. At the entry to a kernel mode routine, the routine should call LIB$LOCK_IMAGE to lock the entire image in the working set. Before exiting the kernel mode routine, the routine should call the LIB$UNLOCK_IMAGE routine.

3.7.9.2 Use of SYS$LCKPAG and SYS$LCKPAG_64

If your application uses the SYS$LCKPAG or SYS$LCKAPG_64 system service to lock code in memory, examine your use of this service. On I64, this service does 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 3.7.9.1). See also the *HP OpenVMS RTL Library (LIB$) Manual* for information about the LIB$LOCK_IMAGE and LIB$UNLOCK_IMAGE routines.

3.7.9.3 Terminal Drivers

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

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

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

3.7.9.4 Protected Image Sections

Protected image sections usually occur in shareable images that 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 I64 have added some subtle restrictions that might require changes in the protected images.

As on VAX and Alpha, data sections that are writable in privileged modes (kernel or exec) can be read by unprivileged (user) mode. The hardware protection for such pages does not allow execute access from any mode. Protected image sections that are linked as both writable and executable are protected to allow inner-mode read, write, and execute; user mode access is not allowed. Because neither user-mode access to inner-mode writable data, nor code being in writable sections, is a common practice, few applications are likely to require both in a single section.

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

4
Overview of Recompiling and Relinking

This chapter introduces the general process of moving an application that runs on a VAX system to an I64 system by recompiling and relinking the source files that make up the application.

In general, if your application is written in a high-level programming language, you should be able to run it on an I64 system with a minimum of effort. High-level languages insulate applications from dependence on the underlying machine architecture. In addition, the programming environment on I64 systems duplicates most of the programming environment on VAX systems. Using native I64 versions of the language compilers and the OpenVMS Linker utility (linker), you can recompile and relink the source files that make up your application to produce a native I64 image.

If your application is written in VAX MACRO, you might be able to run it on an I64 system with minimum effort, although it is more likely to contain some dependencies on the underlying VAX architecture that require intervention.

Privileged applications, which run in inner modes or at elevated interrupt priority levels (IPLs), might require significant changes because of assumptions incorporated in the code about the internal operation of the operating system. Typically, such applications require significant changes after a major release of the OpenVMS VAX operating system.

Note

Remember that it is possible to introduce architectural dependencies even in applications written in high-level languages. In addition, hidden bugs in your application might come to light during the move to a new platform.

4.1 Compiling Applications on VAX With Current Compiler Version

Before recompiling your code with a native I64 compiler, HP recommends that you first compile the code to run on VAX with the latest version of the compiler. This action might uncover problems that are attributable to the change in the compiler version only. For example, newer versions of compilers might enforce programming language standards that were previously ignored, exposing latent problems in your application code. Newer versions might also enforce changes to the standard that were not in effect when the earlier versions were created. Fixing such problems on OpenVMS VAX simplifies porting the application to I64.

4.2 Recompiling Your Application with Native I64 Compilers

Many of the languages supported on VAX systems, such as Fortran and C, are also supported on I64 systems. For information about the compilers for some common programming languages on I64 systems, see Chapter 9.

The compilers available on I64 systems are intended to be compatible with their counterparts on VAX systems. The compilers conform to language standards and include support for most VAX language extensions. The compilers produce output files with the same default file types as they do on VAX systems, such as .OBJ for an object module.

See Section 3.2.1.1 for a list of the OpenVMS I64 compilers.

Note, however, that some features supported by the compilers on VAX systems might not be available in the same compiler on I64 systems. In addition, some compilers on I64 systems support new features not supported by their counterparts on VAX systems. To provide compatibility, some compilers support compatibility modes. For example, the HP C compiler for OpenVMS I64 systems supports a VAX C compatibility mode that is invoked by specifying the /STANDARD=VAXC qualifier.

If the VAX application was built some time ago, especially with old compilers, rebuilding it with the latest compiler versions is a good preparation for the migration. It is more likely that I64 compilers are feature compatible with the latest VAX compilers than with old compilers.

4.3 Relinking Your Application on an I64 System

Once you successfully recompile your source files, you must relink your application to create a native I64 image. The linker produces output files with the same file types as on current VAX systems. For example, by default, the linker uses the file type .EXE to identify image files.

Because the way in which you perform certain linking tasks is different on I64 systems, you probably need to modify the LINK command used to build your application. The following list describes some linker changes that might affect your application's build procedure. See the *HP OpenVMS Linker Utility Manual* for more information.

- **Declaring universal symbols in shareable images**—If your application creates shareable images, your application build procedure probably includes a transfer vector file, written in VAX MACRO, in which you declare the universal symbols in the shareable image. On I64 systems, instead of creating a transfer vector file, you must declare universal symbols in a linker options file by specifying the option SYMBOL_VECTOR=*option*.
- **Linking against the OpenVMS executive**—On VAX systems, you link against the OpenVMS executive by including the system symbol table file (SYS.STB) in your build procedure. On I64 systems, you link against the OpenVMS executive by specifying the /SYSEXE qualifier.
- **Processing shareable images implicitly**— On I64 systems, you must specify any shareable images that your image has direct calls to in your build procedure.

On VAX systems, the linker puts the list of images that the shareable image was linked against into the image's dependency list. For example, if you link a main image against LIBRTL.EXE, and LIBRTL.EXE was linked against LIBOTS.EXE, both LIBRTL and LIBOTS will be in the main image's dependency list (also called the shareable image list) even though the main image has no direct calls to LIBOTS.EXE. If LIBOTS changes in an incompatible way, you must relink the main image.

On I64 systems, the linker only puts the images you linked directly against into an image's dependency list. If on I64 an image is linked against LIBRTL.EXE, and if LIBRTL.EXE was linked against LIBOTS.EXE, only LIBRTL will be in the main image's dependency list. If LIBOTS.EXE changes in an incompatible way, LIBRTL.EXE must be relinked, but the main image is fine.

- **No based clusters**— Specifying a base address in a CLUSTER option is permitted on VAX. But on I64, specifying a base address in a CLUSTER option is illegal. See the description of the CLUSTER= option in Table 4–2.
- **Handling of initialized overlaid program sections is different on OpenVMS I64**— On VAX systems, initializations can be performed on different 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. When an initialization is made on I64 systems, the entire section is initialized by the compiler. Subsequent initializations of this section can be performed only if the new initialization matches the existing one.

The linker supports several qualifiers and options that that are specific to I64 systems. These qualifiers are listed in Table 4–1. Table 4–2 lists linker qualifiers and options supported on VAX systems but not supported on I64 systems. Table 4–3 lists linker qualifiers and options supported on VAX systems but ignored by I64 systems.

Table 4–1 Linker Qualifiers and Options Specific to OpenVMS I64 Systems

Qualifiers	Description
/DEMAND_ZERO[=PER_PAGE]	Enables demand-zero image sections (referred to as segments on OpenVMS I64) for both executable and shareable images. The keyword PER_PAGE directs the linker to compress trailing zeros for each segment (that is, demand-zero compression of zeros on trailing pages.)
/DSF	Directs the linker to create a file called a debug symbol file (DSF) for use by the OpenVMS I64 Debugger.
/FP_MODE=keyword	The OpenVMS 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 does not override an initial floating-point mode provided by the main transfer module.
/FULL[=(*keyword* [,...])]	The keyword GROUP_SECTIONS prints all of the groups that were used in the map. (Groups are a new concept in the object language for OpenVMS I64. For more details, see the *HP OpenVMS Linker Utility Manual*.) The keyword NOSECTION_ DETAILS is specified when the OpenVMS I64 linker suppresses zero length contributions in the Program Section Synopsis of the map. The default for is /FULL=SECTION_DETAILS.
/GST	Directs the linker to create a global symbol table (GST) for a shareable image (the default). Typically specified as /NOGST when used to ship an application with a shareable image that cannot be linked against.
/INFORMATIONALS	Directs the linker to output informational messages during a link operation (the default). More typically specified as /NOINFORMATIONALS to suppress these messages.
/NATIVE_ONLY	Directs the linker to not pass along the procedure signature block (PSB) information (created by the compilers) in the image it is creating (the default). If you specify /NONATIVE_ONLY during linking, the image activator uses the PSB information, if any, provided in the object modules that are specified as input files to the link operation that invoke jacket routines. Jacket routines are necessary to allow native I64 images to work with translated images.
/SEGMENT_ ATTRIBUTE=(*segment_attribute* [, ...])	Instructs the OpenVMS I64 linker to set certain attributes for segments. The OpenVMS I64 Linker accepts DYNAMIC=address_region, SHORT=WRITE, CODE=address_region, and SYMBOL_VECTOR=[NO]SHORT as segment attributes, where an address region can be specified with keywords P0 and P2.

(continued on next page)

Table 4–1 (Cont.) Linker Qualifiers and Options Specific to OpenVMS I64 Systems

Qualifiers	Description
/SYSEXE	Directs the linker to process the OpenVMS executive image (SYS$BASE_IMAGE.EXE) to resolve symbols left unresolved in a link operation.
Options	**Description**
SYMBOL_TABLE=*option*	Directs the linker to include global symbols as well as universal symbols in the symbol table file associated with a shareable image. By default, the linker includes only universal symbols.
SYMBOL_VECTOR=*option*	Used to declare universal symbols in I64 shareable images.

Table 4–2 OpenVMS VAX Linker Qualifiers and Options Not Supported on I64 Systems

Qualifier	Description
/DEBUG=file_spec	Specifying an object file with the /DEBUG qualifier to get the user-written debugger module activated at runtime is no longer allowed.
/SYSTEM[=base_address]	Directs the linker to create a system image and optionally allows you to secify the address at which the image should be loaded into memory. A system image cannot be activated with the RUN command; it must be bootstrapped or otherwise loaded into memory.
Options	**Description**
BASE=*option*	Specifies the base address (starting address) that you want the linker to assign to the image.
CLUSTER=*cluster_name, base_address*	The base address keyword must be null in the CLUSTER option. Based images, or images with based image sections, are not supported by I64.
UNIVERSAL=*option*	Declares a symbol in a shareable image as universal, causing the linker to include it in the GST of a shareable image.

Table 4–3 OpenVMS VAX Linker Qualifiers and Options Ignored on I64 Systems

Qualifier	Description
/ALPHA	Directs the linker to produce an OpenVMS Alpha image.
/HEADER	When specified with the /SYSTEM qualifier, directs the linker to include an image header in a system image. The I64 linker ignores the /HEADER qualifier if it is specified for an image. The VAX and Alpha linkers ignore it whenever it is specified for an executable or shareable image.
/VAX	Directs the linker to produce an OpenVMS VAX image.

Qualifier	Description
DZRO_MIN=*option*	Specifies the minimum number of contiguous, uninitialized pages that the linker must find in an image section before it can extract the pages from the image section and place them in a newly created demand-zero image section. By creating demand-zero image sections (image sections that do not contain initialized data), the linker can reduce the size of images.
ISD_MAX=*option*	Specifies the maximum number of image sections allowed in the image.

4.4 Compatibility Between the Mathematics Libraries Available on VAX and I64 Systems

Mathematical applications using the standard OpenVMS call interface to the OpenVMS Mathematics (MTH$) Run-Time Library need not change their calls to MTH$ routines when migrating to an OpenVMS I64 system. Jacket routines are provided that map MTH$ routines to their `math$` counterparts in the HP Portable Mathematics Library (DPML) for OpenVMS I64 systems. However, there is no support in the DPML for calls made to JSB entry points and vector routines. Note that DPML routines are different from those in the OpenVMS MTH$ RTL; expect to see small differences in the precision of the mathematical results.

To maintain compatibility with future libraries and to create portable mathematical applications, HP recommends using the DPML routines available through the high-level language of your choice (for example, HP C or HP Fortran) rather than using the call interface. DPML routines afford significantly higher performance and accuracy.

See the *Compaq Portable Mathematics Library* manual for more information about DPML routines.

4.5 Determining the Host Architecture

Your application might need to determine whether it is running on an OpenVMS VAX system or an I64 system. From within your program, you can obtain this information by calling the $GETSYI system service (or the LIB$GETSYI RTL routine), and specifying the ARCH_TYPE item code. When your application is running on a VAX system, the $GETSYI system service returns the value 1.

When your application is running on an I64 system, the $GETSYI system service returns the value 3.

Example 4–1 shows how to determine the host architecture in a DCL command procedure by calling the F$GETSYI DCL command and specifying the ARCH_TYPE item code. (For an example of calling the $GETSYI system service to obtain the page size of an I64 system, see Section 5.4.)

Example 4–1 Using the ARCH_TYPE Keyword to Determine Architecture Type

```
$! Determine architecture type
$ type_symbol = f$getsyi("arch_type")
$ if type_symbol .eq. 1 then goto ON_VAX
$ if type_symbol .eq. 2 then goto ON_ALPHA
$ if type_symbol .eq. 3 then goto ON_I64
$ !
$ ! Unknown architecture
$ !
$ exit
$ ON_VAX:
$ !
$ ! Do VAX-specific processing
$ !
$ exit
$ ON_ALPHA:
$ !
$ ! Do Alpha-specific processing
$ !
$ exit
$ ON_I64:
$ !
$ ! Do I64-specific processing
$ !
$ exit
```

Note that the ARCH_TYPE item code is available only on VAX systems running OpenVMS Version 5.5 or later. If your application needs to determine the host architecture for earlier versions of the operating system, use one of the other $GETSYI system service item codes listed in Table 4–4.

Table 4–4 $GETSYI Item Codes That Specify Host Architecture

Keyword	Usage
ARCH_TYPE	Returns 1 on VAX systems; returns 2 on Alpha systems; returns 3 on a I64 systems. Supported on I64 systems, and on VAX systems running OpenVMS Version 5.5 or later.
ARCH_NAME	Returns text string “VAX” on VAX systems, text string “Alpha” on Alpha systems, text string “IA64” on I64 systems. Supported on I64 systems, and on VAX systems running OpenVMS Version 5.5 or later.
HW_MODEL	Returns an integer that identifies a particular hardware model. A value equal to 4096 identifies I64 systems.
CPU	Returns an integer that identifies a particular CPU. The value 128 identifies a system as “not a VAX.” This code is supported on much earlier versions of OpenVMS than the ARCH_TYPE and ARCH_NAME codes.

5
Adapting Applications to a Larger Page Size

This chapter describes how to identify dependencies your application might have on the VAX page size and makes recommendations for correcting those dependencies.

5.1 Overview

In general, page size, the basic unit of memory manipulated by the operating system, is below the level of applications, especially for applications written in high-level or mid-level programming languages. However, your application might contain page-size dependencies if it calls system services or run-time library routines to perform memory management functions such as the following:

- Allocating virtual memory
- Mapping sections into the virtual address space of your process
- Locking memory into your working set
- Protecting segments of your virtual address space

The system services and run-time library routines that perform these functions manipulate memory in pages. The values you specified as arguments to these routines are based on an assumption of a 512-byte page, the page size defined by the VAX architecture. OpenVMS I64 supports an 8-KB default page size, and will support 16-KB, 32-KB, or 64-KB page size in future OpenVMS releases. The following sections provide more information about examining the routines.

Note that this difference in page sizes does not affect memory allocation using higher level routines, for example, the run-time library routines that manipulate virtual memory zones or language-specific memory allocation routines, such as the `malloc` and `free` routines in C.

5.1.1 Compatibility Features

Wherever possible, system services or run-time library routines attempt to present the same interface and return values on I64 systems as they do on VAX systems. For example, on I64 systems, the routines that accept page-count values as arguments still interpret these arguments in 512-byte quantities, called **pagelets** to distinguish them from the CPU-specific page size. The routines convert pagelet values into CPU-specific pages. The routines that return page-count values convert from CPU-specific pages to pagelets so that the return values are still measured in 512-byte units.

Note

On I64 systems, when creating page frame sections you cannot use the $CRMPSC routine with the flag SEC$M_PFNMAP. You must call SYS$CREATE_GPFN, SYS$CRMPSC_GPFN_64, or SYS$CRMPSC_ PFN_64 system services. You must also use the flag SEC$M_ARGS64 to

indicate that you have provided a 64-bit **start_pfn** argument. Note that 64-bit system services can be called with sign-extended 32-bit addresses.

5.1.2 Summary of Memory Management Routines with Potential Page-Size Dependencies

Despite the compatibility, some routines behave differently on I64 systems than they do on VAX systems and might require you to modify your source code. For example, on I64 systems, the system services that map section files ($CRMPSC and $MGBLSC) require you to specify address value arguments that are aligned on CPU-specific page boundaries. On VAX systems, these routines round the address values specified in arguments to VAX page boundaries. On I64 systems, the routines do not round these addresses to CPU-specific page boundaries.

Table 5–1 lists the memory management routines with the arguments they support that may contain page-size dependencies. The table lists the arguments with their intended function and describes how these arguments are interpreted on I64 systems. Note that the table does not attempt to list all the arguments accepted by each routine. For more information about the routines and their argument lists, see the *HP OpenVMS System Services Reference Manual*.

Table 5–1 Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on I64 Systems
Adjust Working Set Limit ($ADJWSL)	**pagcnt** specifies the number of pages to add to (or subtract from) the current working set limit.	Interpreted in pagelets, adjusted up or down to represent CPU-specific-sized pages.
	wsetlm specifies the value of the current working set limit.	Interpreted in pagelets, adjusted up or down to represent CPU-specific-sized pages.
Create Process ($CREPRC)	**quota** accepts several quota descriptors that specify page counts, such as the default working set size, paging file quota, and working set expansion quota.	Interpreted in pagelets, adjusted up or down to represent CPU-specific-sized pages.
Create Virtual Address ($CRETVA)	**inadr** specifies the start- and end-addresses of the memory to be allocated. If the end-address is the same as the start-address, a single page is allocated.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory affected by the call.	Unchanged.

(continued on next page)

Table 5–1 (Cont.) Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on I64 Systems
Create and Map Section ($CRMPSC)	**inadr** specifies the start- and end-addresses that define the region to be remapped. If the end-address is the same as the start-address, a single page is mapped unless the SEC$M_EXPREG flag is set, in which case the start-address is interpreted as determining whether the allocation should be in P0 or P1 space.	Addresses must be aligned on CPU-specific pages (unless the SEC$M_EXPREG flag is set); no rounding is done. (See Section 5.3 for more information about mapping.)
	retadr specifies the actual start- and end-addresses of the memory affected by the call.	Returns the start- and end-addresses of the *usable* range of addresses, which might be different than the total amount mapped. This argument is required when the **relpag** argument is specified.
	flags specifies the type and characteristics of the section to be created or mapped.	The flag bit SEC$M_NO_OVERMAP indicates that existing address space should not be overmapped.
	relpag specifies the page offset at which mapping of the section file should begin.	Interpreted as an index into the section file; measured in pagelets.
	pagcnt specifies the number of pages (blocks) in the file to be mapped.	Interpreted in pagelets; no rounding is done.
	pfc specifies the number of pages that should be mapped when a page fault occurs.	Interpreted in CPU-specific-sized pages. When specifying a value for this argument, remember that, because I64 systems support 8-K, 16-K, 32-K, and 64-K byte physical page sizes, at least 16 pagelets are mapped for each physical page. The system cannot map less than a physical page.
Delete Virtual Address ($DELTVA)	**inadr** specifies the start- and end-addresses of the memory to be deallocated.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was deleted.	Unchanged.
Expand Program/Control Region ($EXPREG)	**pagcnt** specifies the amount of memory to allocate, in 512-byte units.	Interpreted in pagelets.
	retadr specifies the actual start- and end-addresses of the memory affected by the call.	Unchanged.

(continued on next page)

Table 5–1 (Cont.) Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on I64 Systems
Get Job/Process Information ($GETJPI)	**itmlst** specifies which information about the process is to be returned.	Many items, such as JPI$_WSEXTENT, interpreted as pagelet values. See the *HP OpenVMS System Services Reference Manual* for more information.
Get Queue Information ($GETQUI)	**itmlst** specifies information to be used in performing the function specified by the **func** argument.	Several items interpreted as pagelet values. See the *HP OpenVMS System Services Reference Manual* for more information.
Get Systemwide Information ($GETSYI)	**itmlst** specifies which information is to be returned about the node or nodes.	Several items interpreted as pagelet values. One additional item, SYI$_PAGE_SIZE, specifies the page size supported by the node. See the *HP OpenVMS System Services Reference Manual* for more information.
Get User Authorization Information ($GETUAI)	**itmlst** specifies which information from the user's user authorization file is to be returned.	Several items return pagelet values. See the *HP OpenVMS System Services Reference Manual* for more information.
Lock Page ($LCKPAG)	**inadr** specifies the start- and end-addresses of the memory to be locked.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was locked.	Unchanged.
Lock Image in Process Working Set ($LOCK_IMAGE)	**address** specifies the address of a byte within the image to be locked in the working set.	Available on VAX and I64 only.
Map Global Section ($MGBLSC)	**inadr** specifies the start- and end-addresses that define the region to be remapped. If the end-address is the same as the start-address, a single page is mapped, unless the SEC$M_EXPREG flag is set, in which case the start-address is interpreted as determining whether the allocation should be in P0 or P1 space.	Addresses must be aligned on a CPU-specific page (unless the SEC$M_EXPREG flag is set); no rounding is done. (See Section 5.3 for more information about mapping.)
	retadr specifies the actual start- and end-addresses of the memory affected by the call.	Returns start- and end-addresses of *usable* portion of memory mapped.
	relpag specifies the page offset at which mapping of the section file should begin.	Interpreted as an index into the section file, measured in pagelets.
Purge Working Set ($PURGWS)	**inadr** specifies the start- and end-addresses of the memory to be purged.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.

(continued on next page)

Table 5–1 (Cont.) Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on I64 Systems
Set Protection ($SETPRT)	**inadr** specifies the start- and end-addresses of the memory to be protected.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was protected.	Unchanged.
Set User Authorization File ($SETUAI)	**itmlst** specifies which information from the user authorization file is to be set.	Several items interpreted in pagelet values. See the *HP OpenVMS System Services Reference Manual* for more information.
Send to Job Controller ($SNDJBC)	**itmlst** specifies information to be used in performing the function specified by the **func** argument.	Several items interpreted in pagelet values. See the *HP OpenVMS System Services Reference Manual* for more information.
Unlock Page ($ULKPAG)	**inadr** specifies the start- and end-addresses of the memory to be unlocked.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was unlocked.	Unchanged.
Unlock Image from Process Working Set ($UNLOCK_IMAGE)	**address** specifies the address of a byte within the image to be unlocked in the working set.	Available on VAX and I64 only.
	retadr specifies the actual start- and end-addresses of the memory that was unlocked.	Unchanged.
Update Section ($UPDSEC)	**inadr** specifies the start- and end-address of the section to write to disk.	Rounds requests to CPU-specific pages. Note that only the address range actually represented by on-disk storage is written to disk.
	retadr specifies the actual start- and end-addresses of the memory that was written to disk.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.

The run-time library routines listed in Table 5–2 allocate (or free) pages of memory. For compatibility, these routines also interpret the page-count information you specify in pagelets.

Table 5–2 Potential Page-Size Dependencies in Run-Time Library Routines

Routine	Argument	Behavior on I64 Systems
LIB$GET_VM_PAGE	**number-of-pages** argument specifies the number of contiguous pages to allocate.	Interpreted in pagelets, rounded to CPU-specific pages.
LIB$FREE_VM_PAGE	**number-of-pages** argument specifies the number of contiguous pages to free.	Interpreted in pagelets, rounded to CPU-specific pages.

5.2 Examining Memory Allocation Routines

To determine whether the memory allocation performed by your application requires modification, check to see where the memory is allocated. The system service routines that perform memory allocation ($EXPREG and $CRETVA) allow you to allocate memory in two ways:

- By expanding the size of the P0 or P1 regions of your application's virtual address space
- By reclaiming a region of your application's existing virtual address space, starting at a location you specify

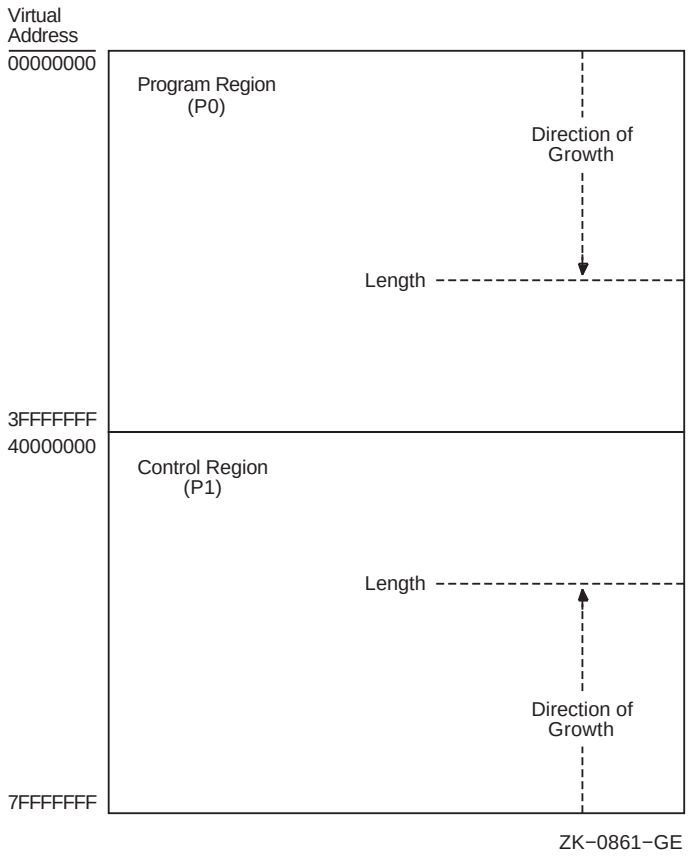
The Intel Itanium architecture defines the same virtual address space layout as the VAX architecture and allows for growth of the P0 and P1 regions in the same direction as on VAX systems. Figure 5–1 shows this layout.

5.2.1 Allocating Memory in Expanded Virtual Address Space

If your application allocates memory by expanding virtual address space using the $EXPREG system service, source code changes might be unnecessary because the values you specified as arguments are valid on I64 systems and VAX systems. The reasons for this are as follows:

- On I64 systems, the $EXPREG system service interprets the amount of memory requested (specified as a page count in the **pagcnt** argument) in 512-byte units, the same as on an VAX system. Thus, the value your application specified still requests the same amount of memory. Note, however, that because the system service rounds the value up to CPU-specific pages, the actual amount of memory allocated by the system for your application might be larger on an I64 system than it is on a VAX system. The entire amount of memory allocated is available for use by your application. Because applications typically allocate memory to satisfy buffer requirements (which do not change with different platforms) the value you specified should still satisfy the requirements of your application.
- Because the allocation occurs in an expanded area of virtual address space, the discrepancy between the amount requested and the amount actually allocated by the system should have no effect on the function of your application.

Figure 5–1 Virtual Address Layout



Recommendation

Your application might not need to be modified. However, HP suggests that you obtain the exact boundaries of the memory allocated by the system, because the amount of memory returned by the $EXPREG system service can vary among implementations of the Intel Itanium architecture. To do this, specify the optional **retadr** argument to the $EXPREG system service, if your application does not already include it. The **retadr** argument contains the start-address and the end-address of the memory allocated by the system service.

For example, the program in Example 5–1 calls the $EXPREG system service to request 10 additional pages of memory. If you run this program on a VAX system, the $EXPREG system service allocates 5120 bytes of additional memory. If you run this program on an I64 system, the $EXPREG system service allocates at least 8192 bytes and possibly more, depending on the page size of the particular implementation of the Intel Itanium architecture.

Example 5–1 Allocating Memory by Expanding Your Virtual Address Space

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
#include  <dvidef.h>

#define  PAGE_COUNT 10  ❶
#define  P0_SPACE   0
#define  P1_SPACE   1

main( argc, argv )
int argc;
char *argv[];
{
     int    status = 0;
     long   bytes_allocated, addr_returned[2];

❷    status = SYS$EXPREG( PAGE_COUNT, &addr_returned, 0, P0_SPACE);

     bytes_allocated = addr_returned[1] - addr_returned[0];

     if( status == SS$_NORMAL)
        printf("bytes allocated = %d\n", bytes_allocated );
     else
        return (status);

}
```

The following items correspond to Example 5–1:

❶ The program defines a symbol, PAGE_COUNT, to stand for the number of pages requested.

❷ The program requests 10 additional pages to be added at the end of the P0 region of its virtual address space.

5.2.2 Allocating Memory in Existing Virtual Address Space

If your application reallocates memory that is already in its virtual address space by using the $CRETVA system service, you might need to modify the values of the following arguments to $CRETVA:

- If your application explicitly rounds the address specified in the **inadr** argument to be a multiple of 512 in order to align on a VAX page boundary, you need to modify the address. On I64 systems, the $CRETVA system service rounds the start-address down to a CPU-specific page boundary, which vary with different implementations.
- The size of the reallocation, specified by the address range in the **inadr** argument, might be larger on an I64 system than it is on a VAX system because the request is rounded up to CPU-specific pages. This can cause the unintended destruction of neighboring data, which also occurs with single-page allocations. (When the start-address and the end-address specified in the **inadr** argument match, a single page is allocated.)

Recommendations

To determine whether your application needs to be modified, HP suggests doing the following:

- For all potential page sizes, make sure the area of virtual address space affected by the call does not destroy important data.

- For all potential page sizes, make sure the start-address at which the allocation begins always falls on a page boundary.
- Specify the optional **retadr** argument, if not already included by your application, to determine the exact boundaries of the memory allocated by the call to the $CRETVA system service.

Example 5–2 shows how memory allocated to a buffer can be reallocated by using the $CRETVA system service.

Example 5–2 Allocating Memory in Existing Address Space

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
#include  <dvidef.h>

char _align(page) buffer[1024];

main( argc, argv )
int argc;
char *argv[];
{

     int      status = 0;
     long     inadr[2];
     long     retadr[2];

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[1023];

     printf("inadr[0]=%u,inadr[1]=%u\n",inadr[0],inadr[1]);

     status = SYS$CRETVA(inadr, &retadr, 0);

     if( status & STS$M_SUCCESS )
     {
        printf("success\n");
        printf("retadr[0]=%u,retadr[1]=%u\n",retadr[0],retadr[1]);
     }
     else
     {
        printf("failure\n");
        exit(status);
     }
}
```

5.2.3 Deleting Virtual Memory

Calls to the $DELTVA system service to free memory allocated by the $EXPREG and $CRETVA system services should require no modification if your application uses the address range returned in the **retadr** argument (returned by the routine used to allocate the memory) as the **inadr** argument to the $DELTVA system service. Because the actual amount of the allocation varies with the implementation, your application should not make any assumptions regarding the extent of the allocation.

5.3 Examining Memory Mapping Routines

To determine whether the memory mapping performed by your application requires modification, check to see where in virtual memory your application performs the mapping. The memory mapping system services ($CRMPSC and $MGBLSC) allow you to map memory in the following ways:

- Map memory into an expanded area of your application's virtual address space.
- Map a single page of memory into your application's virtual address space, starting at a location you specify (the location might be in existing virtual address space).
- Map memory into an existing area of your virtual address space, defined by the start-address and end-address you specify.

How your application maps a section is determined primarily by the following arguments to the $CRMPSC and $MGBLSC system services:

- **inadr** argument—Specifies the size and location of the section by its start-address and end-address, interpreted by the $CRMPSC system service in the following ways:
  - If both addresses specified in the **inadr** argument are the same and the SEC$M_EXPREG bit is set in the **flags** argument, the system service allocates the memory in whichever program region the addresses fall, but it does not use the specified location.
  - If both addresses specified in the **inadr** argument are the same and the SEC$M_EXPREG flag is not set, a single page is mapped, starting at the specified location. (This mode of operation of the $CRMPSC system service is not supported on I64 systems. If your application uses this mode, see Section 5.3.2 for recommendations about modifying your source code.)
  - If both addresses are different, the system service maps the section into memory using the boundaries specified.
- **pagcnt** (page count) argument—Specifies the number of blocks you want to map from the section file.
- **relpag** (relative page number) argument—Specifies the location in the section file at which you want mapping to begin.

The $CRMPSC and $MGBLSC system services map a minimum of one CPU-specific page. If the section file does not fill a single page, the remainder of the page is filled with zeros. The extra space on the page should not be used by your application because only the data that fits into the section file will be written back to the disk.

5.3.1 Mapping into Expanded Virtual Address Space

If your application maps a section file into an expanded area of your application's virtual address space, you might not need to modify the source code. Because the mapping occurs in expanded virtual address space, there is no danger of overmapping existing data, even if the amount of memory allocated is larger on an I64 system than on a VAX system. Thus, the values you specify as arguments to the $CRMPSC system service on a VAX system should still work on an I64 system.

Recommendation

While applications that map sections into expanded areas of virtual memory may work correctly without modification, HP suggests that you specify the **retadr** argument, if not already specified by your application, to determine the exact boundaries of the memory that was mapped by the call.

Note

If your application specifies the **relpag** argument, you must specify the **retadr** argument; it is not an optional argument. For more information about using the **relpag** argument, see Section 5.3.4.

Example 5–3 shows a call to the $CRMPSC system service that maps a section file into expanded address space. The example maps a section file named MAPTEST.DAT that was created using the DCL command CREATE. For example:

```
$  CREATE maptest.dat
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
Ctrl/Z
```

Example 5–3 Mapping a Section into Expanded Virtual Address Space

```
#include  <ssdef.h>
#include  <string.h>
#include  <stdlib.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
#include  <dvidef.h>
#include  <rms.h>
#include  <secdef.h>

struct FAB fab;

char _align(page) buffer[1024];
char *filename = "maptest.dat";

main( argc, argv )
int argc;
char *argv[];
{

     int    status = 0;
     long   flags = SEC$M_EXPREG;
     long   inadr[2];
     long   retadr[2];
     int    fileChannel;
```

(continued on next page)

Example 5–3 (Cont.) Mapping a Section into Expanded Virtual Address Space

```
/********  create disk file to be mapped *************/

     fab = cc$rms_fab;
     fab.fab$l_fna = filename;
     fab.fab$b_fns = strlen( filename );
     fab.fab$l_fop = FAB$M_CIF | FAB$M_UFO;  /* must be UFO */

     status = sys$create( &fab );

     if( status & STS$M_SUCCESS )
        printf("%s opened\n",filename);
     else
     {
        exit( status );
     }

     fileChannel = fab.fab$l_stv;

/**********  create and map the section  ****************/

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[0];

     status = SYS$CRMPSC( inadr, /* inadr=address target for map */
                        &retadr, /* retadr= what was actually mapped */
                              0, /* acmode  */
                          flags, /* flags, with SEC$M_EXPREG bit set */
                              0, /* gsdnam, only for global sections */
                              0, /* ident, only for global sections */
                              0, /* relpag, only for global sections */
                    fileChannel, /* returned by SYS$CREATE */
                              0, /* pagcnt = size of sect. file used */
                              0, /* vbn = first block of file used */
                              0, /* prot = default okay */
                              0); /* page fault cluster size */

     if( status & STS$M_SUCCESS )
     {
          printf("section mapped\n");
          printf("retadr[0]=%u,retadr[1]=%u\n",retadr[0],retadr[1]);
     }
     else
     {
          printf("map failed\n");
          exit( status );
     }

}
```

5.3.2 Mapping a Single Page to a Specific Location

If your application maps a section file into a single page of memory, you need to modify your source code because this mode of operation is not supported on I64 systems. Because the page size on I64 systems differs from that on VAX systems and varies with different implementations of the Intel Itanium architecture, you must specify the exact boundaries of the memory into which you intend to map a section file. The $CRMPSC system service returns an invalid arguments error (SS$_INVARG) for this usage.

To see whether your application uses this mode, check the start-address and end-address specified in the **inadr** argument. If both addresses are the same and the SEC$M_EXPREG bit in the **flags** argument is not set, your application is using this mode.

Recommendations

HP suggests the following guidelines for modifying calls to the $CRMPSC system service in this mode:

- If the location into which the mapping occurs is unimportant, set the SEC$M_EXPREG bit in the **flags** argument and let the system service map the section into an expanded area of your application's virtual address space. For more information about this mode of operation, see Section 5.3.1.
- If the location into which the mapping occurs is important, define both the start-address and end-address in the **inadr** argument and map the section into a defined area. For more information about this mode, see Section 5.3.3.

5.3.3 Mapping into a Defined Address Range

If your application maps a section into a defined area of its virtual address space, you might need to modify your source code because the $CRMPSC and $MGBLSC system services interpret some of the arguments differently on I64 systems than on VAX systems. The differences are as follows:

- The start-address specified in the **inadr** argument must be aligned on a CPU-specific page boundary and the end-address specified must be aligned with the end of a CPU-specific page. On VAX systems, the $CRMPSC and the $MGBLSC system services round these addresses to page boundaries. On I64 systems, which have larger page sizes, automatic rounding is not done because rounding to CPU-specific page boundaries affects a much larger portion of memory. Thus, on I64 systems, you must explicitly state where you want the virtual memory space mapped. If the addresses you specify are not aligned on CPU-specific page boundaries, the $CRMPSC system service returns an invalid arguments error (SS$_INVARG).
- The addresses returned in the **retadr** argument reflect only the usable portion of the actual memory mapped by the call, not the entire amount mapped. The usable amount is either the value specified in the **pagcnt** argument (measured in pagelets) or the size of the section file, whichever is smaller. The actual amount mapped depends on how many CPU-specific pages are required to map the section file. If the section file does not fill a CPU-specific page, the remainder of the page is filled with zeros. The excess space on this page should not be used by your application. The end-address specified in the **retadr** argument specifies the upper limit available to your application. Note also that, when the **relpag** argument is specified, you must also include the **retadr** argument; it is not an optional argument on I64 systems as it is on VAX systems. See Section 5.3.4 for more information.

Recommendations

HP suggests that you change your application so that it maps data into expanded virtual address space, if possible. If you cannot change the way your application maps data, HP recommends the following guidelines:

- Because the operating system maps a minimum of one physical page, and physical pages on I64 systems are larger than pages on VAX systems, you must make sure that, when the system maps the section into the buffer you define in your application, it does not overwrite neighboring data. Most applications on VAX systems define the buffer into which the section is to be mapped in multiples of 512 bytes because that is the page size on VAX systems, even if the section file to be mapped is less than 512 bytes in size. To follow this strategy on I64 systems, you would need to declare a buffer in

your application as large as the largest possible I64 page (64-K bytes) which would waste memory.

A better way to make sure your section does not overwrite neighboring data when it is mapped is to force the linker to isolate the buffer into a separate image section. (The linker creates an image out of image sections. Each image section defines the memory requirements of part of the image.) By isolating the buffer into its own image section, you ensure that the mapping operation does not overwrite neighboring data because the linker allocates image sections on page boundaries; neighboring data starts on the next page boundary. Thus, you can map a page of memory into your section without disturbing neighboring data and without having to change the size of the buffer.

To ensure that the linker puts your section into its own image section, you must set the SOLITARY attribute of the program section in which your section resides, using the linker's PSECT_ATTR= option. (For more information, see the *HP OpenVMS Linker Utility Manual*.) Note that you might need to use the capabilities of whatever high-level or mid-level programming language you are using to ensure that the compiler puts the buffer you define into a separate program section. See compiler documentation for more information.

- Make sure that the start-address and end-address of the section that you specify as arguments to the $CRMPSC and $MGBLSC system services are aligned with the start-address and end-address of a CPU-specific page. On VAX systems, the system services round the addresses to page boundaries for you. On I64 systems, the system services do not round the addresses you specify to page boundaries.

  If you isolate the section into its own image section, using the SOLITARY program section attribute, the start-address is guaranteed to be on a page boundary because the linker aligns image sections on page boundaries by default, no matter what the page size of the host machine is at run time.

  To make sure the end-address of the section is aligned on a CPU-specific page boundary, you must know the page size supported by the machine on which your application is being run. You can obtain the CPU-specific page size at run time by calling the $GETSYI system service or the LIB$GETSYI run-time library routine, and use this value to calculate an aligned end-address value to pass in the **inadr** argument to the system services.

  Note that you should specify the **retadr** argument to determine the amount of usable memory the system mapped. The operating system maps a minimum of one page; however, your application may use only part of the page. The end-address specified in the **retadr** argument marks the upper limit of usable memory. (On I64 systems, if your application specifies the **relpag** argument to the $CRMPSC system service, the **retadr** argument is required.)

The example VAX program in Example 5–4 maps the section file created in Section 5.3.1 into its existing virtual address space. The application defines a buffer, named `buffer`, that is 512 bytes in size, reflecting the VAX page size. The program defines the exact bounds of the section by passing the address of the first byte of the buffer as the start-address and the address of the last byte of the buffer as the end-address in the **inadr** argument.

Example 5–4 Mapping a Section into a Defined Area of Virtual Address Space

```
#include <ssdef.h>
#include <stdio.h>
#include <stsdef.h>
#include <descrip.h>
#include <dvidef.h>
#include <rms.h>
#include <secdef.h>

struct FAB fab;

char *filename = "maptest.dat";

char _align(page) buffer[512];

main( argc, argv )
int argc;
char *argv[];
{

     int    status = 0;
     long   flags = 0;
     long   inadr[2];
     long   retadr[2];
     int    fileChannel;

/********  create disk file to be mapped *************/

     fab = cc$rms_fab;
     fab.fab$l_fna = filename;
     fab.fab$b_fns = strlen( filename );
     fab.fab$l_fop = FAB$M_CIF | FAB$M_UFO;  /* must be UFO */

     status = sys$create( &fab );

     if( status & STS$M_SUCCESS )
        printf("Opened mapfile %s\n",filename);
     else
     {
        printf("Cannot open mapfile %s\n",filename);
        exit( status );
     }

     fileChannel = fab.fab$l_stv;

/**********  create and map the section  ***************/

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[511];

     printf("inadr[0]=%u,inadr[1]=%u\n",inadr[0],inadr[1]);

     status = SYS$CRMPSC(inadr, /* inadr=address target for map */
                         &retadr, /* retadr= what was actually mapped */
                               0, /* acmode  */
                               0, /* flags */
                               0, /* gsdnam, only for global sections */
                               0, /* ident, only for global sections */
                               0, /* relpag, only for global sections */
                     fileChannel, /* returned by SYS$CREATE */
                               0, /* pagcnt = size of sect. file used */
                               0, /* vbn = first block of file used */
                               0, /* prot = default okay */
                               0 ); /* page fault cluster size */
```

(continued on next page)

Example 5–4 (Cont.) Mapping a Section into a Defined Area of Virtual Address Space

```
    if( status & STS$M_SUCCESS )
    {
          printf("Map succeeded\n");
          printf("retadr[0]=%u,retadr[1]=%u\n",retadr[0],retadr[1]);
    }
    else
    {
          printf("Map failed\n");
          exit( status );
    }
}
```

For the program in Example 5–4 to run correctly on an I64 system, you must make the following modifications:

- You must ensure that the start-address of the section specified in the **inadr** argument is aligned on an I64 page boundary and the end-address specified is aligned with the end of an I64 page.
- You must ensure that when a larger page on an I64 system is mapped, neighboring data is not overwritten.

One way to accomplish these goals is to isolate the program section that contains the section data in its own image section by using the SOLITARY program section attribute.

In the example, the section, named `buffer`, appears in the program section named buffer. (Program section creation differs among programming languages on each platform. Check the compiler documentation to ensure that the section is placed in its own program section.), The following link operation illustrates how to set the solitary attribute of this program section:

```
$  LINK MAPTEST, SYS$INPUT/OPT
PSECT_ATTR=BUFFER,SOLITARY
Ctrl/Z
```

To specify an end-address for the section buffer that is aligned with the end of a CPU-specific page boundary, obtain the CPU-specific page size at run time, subtract 1 from the returned value, and use the resulting value to take the address of the last element of the array. Pass this value as the second longword in the **inadr** argument. (To find out how to obtain the page size at run time, see Section 5.4.) Note that you do not need to change the allocation of the buffer into which the section is mapped.

To ensure that your application runs on an I64 system with any page size, specify the /BPAGE=16 qualifier to force the linker to align image sections on 64KB boundaries. Note that the total amount of memory mapped might be much larger than the total amount of usable memory. The amount of usable memory is determined by the value of the page count argument (**pagcnt**) or the size of the section file, whichever is smaller. To avoid using memory that is not within the bounds of the section, use the values returned in the **retadr** argument.

Example 5–5 shows the source changes required for Example 5–4 in in order for it to run on an I64 system.

Example 5–5 Source Code Changes Required to Run Example 5–4 on an I64 System

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <string.h>
#include  <stdlib.h>
#include  <descrip.h>
#include  <dvidef.h>
#include  <rms.h>
#include  <secdef.h>
#include  <syidef.h> ❶

char buffer[512];  ❷
char *filename = "maptest.dat";
struct FAB fab;

long  cpu_pagesize; ❸

struct itm {                        /* item list */
    short int     buflen;  /* length of buffer in bytes */
    short int  item_code;  /* symbolic item code */
    long          bufadr;  /* address of return value buffer */
    long       retlenadr;  /* address of return value buffer length */
  } itmlst[2]; ❹

main( argc, argv )
int argc;
char *argv[];
{
     int    i;
     int    status = 0;
     long   flags = SEC$M_EXPREG;
     long   inadr[2];
     long   retadr[2];
     int    fileChannel;
     char  *mapped_section;

/********  create disk file to be mapped *************/

     fab = cc$rms_fab;
     fab.fab$l_fna = filename;
     fab.fab$b_fns = strlen( filename );
     fab.fab$l_fop = FAB$M_CIF | FAB$M_UFO;  /* must be UFO */

     status = sys$create( &fab );

     if( status & STS$M_SUCCESS )
        printf("%s opened\n",filename);
     else
     {
        exit( status );
     }

     fileChannel = fab.fab$l_stv;
```

(continued on next page)

Example 5–5 (Cont.) Source Code Changes Required to Run Example 5–4 on an I64 System

```
/**********  obtain the page size at run time  ****************/

     itmlst[0].buflen =  4;
     itmlst[0].item_code = SYI$_PAGE_SIZE;
     itmlst[0].bufadr =  &cpu_pagesize;
     itmlst[0].retlenadr = &cpu_pagesize_len;
     itmlst[1].buflen = 0;
     itmlst[1].item_code = 0;

❺    status = sys$getsyiw( 0, 0, 0, &itmlst, 0, 0, 0 );

     if( status & STS$M_SUCCESS )
     {
          printf("getsyi succeeds, page size = %d\n",cpu_pagesize);
     }
     else
     {
          printf("getsyi fails\n");
          exit( status );
     }

/**********  create and map the section  ****************/

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[cpu_pagesize - 1]; ❻

     printf("address of buffer = %u\n", inadr[0] );

     status = SYS$CRMPSC(&inadr, /* inadr=address target for map */
                         &retadr, /* retadr= what was actually mapped */
                               0, /* acmode  */
                               0, /* no flags to set  */
                               0, /* gsdnam, only for global sections */
                               0, /* ident, only for global sections */
                               0, /* relpag, only for global sections */
                     fileChannel, /* returned by SYS$CREATE */
                               0, /* pagcnt = size of sect. file used */
                               0, /* vbn = first block of file used */
                               0, /* prot = default okay */
                               0); /* page fault cluster size */

     if( status & STS$M_SUCCESS )
     {
          printf("section mapped\n");
          printf("start address returned =%u\n",retadr[0]);
     }
     else
     {
          printf("map failed\n");
          exit( status );
     }
}
```

The following items correspond to Example 5–5:

❶ The header file SYIDEF.H contains definitions of OpenVMS item codes for the $GETSYI system service.

❷ The buffer is defined without using the _align(page) storage descriptor. Because the page size cannot be determined until run time on OpenVMS I64 systems, the HP C for OpenVMS I64 compiler aligns the data on the largest I64 page size (64 KB) when _align(page) is specified.

❸ This structure defines the item list used to obtain the page size at run time.

❹ This variable holds the page-size value returned.

❺ This call to the $GETSYI system service obtains the page size at run time.

❻ The end-address of the buffer is specified by subtracting 1 from the page-size value returned.

5.3.4 Mapping from an Offset into a Section File

Your application might map a portion of a section file by specifying the address at which to start the mapping as an offset from the beginning of the section file. You specify this offset by supplying a value to the **relpag** argument of the $CRMPSC system service. The value of the **relpag** argument specifies the page number relative to the beginning of the file at which the mapping should begin.

To preserve compatibility, the $CRMPSC system service interprets the value of the **relpag** argument in 512-byte units on both VAX systems and I64 systems. However, because the CPU-specific page size on I64 systems is larger than 512 bytes, the address specified by the offset in the **relpag** argument probably does not fall on a CPU-specific page boundary. The $CRMPSC system service can map virtual memory in CPU-specific page increments only. Thus, on I64 systems, the mapping of the section file starts at the beginning of the CPU-specific page that contains the offset address, not at the address specified by the offset.

Note

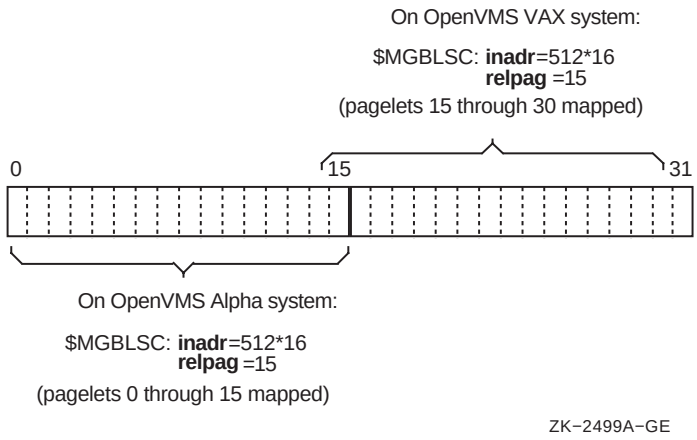
Even though the routine starts mapping at the beginning of the CPU-specific page that contains the address specified by the offset, the start-address returned in the **retadr** argument is the address specified by the offset, not the address at which mapping actually starts.

If your application maps from an offset into a section file, you might need to enlarge the size of the address range specified in the **inadr** argument to accommodate the extra virtual memory space that gets mapped on I64 systems. If the address range specified is too small, your application might not map the entire portion of the section file you want, because the mapping begins at an earlier start-address in the section file.

For example, to map 16 blocks in a section file starting at block number 15 on a VAX system, you could specify an address range 16*512 bytes in size in the **inadr** argument and specify a value of 15 for the **relpag** argument. To accomplish this same mapping on an I64 system, you must allow for the difference in page sizes. For example, on an I64 system with an 8-KB page size, the address specified by the **relpag** offset might fall 15 pagelets into a CPU-specific page, as shown in Figure 5–2. Because the $CRMPSC system service on an I64 system begins mapping of the section file at a CPU-specific page boundary, it would fail to map blocks 16 through 30. For the mapping to succeed, you must increase the size of the address range to accommodate the additional 15 pagelets mapped by the $CRMPSC system service (or the $MGBLSC system service) on an I64

system. Otherwise, only one block of the portion of the section file you specified is mapped.

Figure 5–2 Effect of Address Range on Mapping from an Offset



When trying to calculate how much to enlarge the size of the address range specified in the **relpag** argument, the following formula can be helpful. The formula calculates the maximum number of CPU-specific pages needed to map a given number of pagelets.

$$\frac{(number\_of\_pagelets\_to\_map + (2 * pagelets\_per\_page) - 2)}{pagelets\_per\_page}$$

For example, this formula can be used to calculate how much to enlarge the address range specified in the previous scenario. In the following equation, the page size is assumed to be 8K, so *pagelets_per_page* equals 16:

```
16+((2x16)-2)/16=2.87...
```

Rounding the result down to the nearest whole number, the formula indicates that the address range specified in the **inadr** argument must encompass two CPU-specific pages.

5.4 Obtaining the Page Size at Run Time

To obtain the page size supported by an I64 system, use the $GETSYI system service. Example 5–6 shows how to use this system service to obtain the page size at run time.

Example 5–6 Using the $GETSYI System Service to Obtain the CPU-Specific Page Size

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
```

(continued on next page)

Example 5–6 (Cont.) Using the $GETSYI System Service to Obtain the CPU-Specific Page Size

```
#include  <dvidef.h>
#include  <rms.h>
#include  <secdef.h>
#include  <syidef.h>  /* defines page size item code symbol */

struct itm {              /* define item list                        */
    short int     buflen;  /* length in bytes of return value buffer */
    short int  item_code;  /* item code                               */
    long          bufadr;  /* address of return value buffer         */
    long       retlenadr;  /* address of return value length buffer  */
  } itmlst[2];

long  cpu_pagesize;
long  cpu_pagesize_len;

main( argc, argv )
int argc;
char *argv[];
{
     int    status = 0;

     itmlst[0].buflen =  4;               /*  page size requires 4 bytes  */
     itmlst[0].item_code = SYI$_PAGE_SIZE; /*  page size item code         */
     itmlst[0].bufadr =  &cpu_pagesize;    /*  address of ret_val buffer   */
     itmlst[0].retlenadr = &cpu_pagesize_len; /* addr of length of ret_val */
     itmlst[1].buflen = 0;
     itmlst[1].item_code = 0;   /* Terminate item list with longword of 0  */

     status = sys$getsyiw( 0, 0, 0, &itmlst, 0, 0, 0 );

     if( status & STS$M_SUCCESS )
     {
          printf("getsyi succeeds, page size = %d\n",cpu_pagesize);
          exit( status );
     }
     else
     {
          printf("getsyi fails\n");
          exit( status );
     }
}
```

5.5 Locking Memory in the Working Set

The $LKWSET system service locks into the working set the range of pages identified in the **inadr** argument as an address range on VAX systems. The system service rounds the addresses to CPU-specific page boundaries if necessary.

A newer LIBRTL routine called LIB$LOCK_IMAGE provides similar functionality on OpenVMS Alpha and I64. LIB$LOCK_IMAGE and the related routine LIB$UNLOCK_IMAGE are preferable for locking code and related data in the working set. For more information about locking images in the working set, see the descriptions of these routines in the *HP OpenVMS RTL Library (LIB$) Manual*.

6
Preserving the Integrity of Shared Data

This chapter describes synchronization mechanisms that ensure the integrity of shared data, such as the atomicity guaranteed by certain VAX instructions.

6.1 Overview

If your application uses multiple threads of execution and the threads share access to data, you might need to add explicit synchronization mechanisms to your application to protect the integrity of the shared data on I64 systems. Without synchronization, an access to the data initiated by one application thread can interfere with an access initiated simultaneously by a competing thread, leaving the data in an unpredictable state.

On VAX systems, the degree of synchronization required depends on the relationship of the different threads of execution, which can include the following:

- Multiple threads executing within a single process, such as a main thread interrupted by an asynchronous system trap (AST) thread.

  Note that the AST thread can be initiated either by the application or by the operating system. For example, the operating system uses an AST to write status to an I/O status block. The operating system also uses an AST to complete a buffered I/O read operation to a specified user buffer.

- Multiple threads separated into multiple processes executing on a single processor that access a global section.

- Multiple threads separated into multiple processes executing *concurrently* on multiple processors that access a global section.

On VAX systems, applications that take advantage of the parallel processing potential of a multiprocessor system have always had to provide explicit synchronization mechanisms, such as locks, semaphores, and interlocked instructions, to protect shared data. However, applications that use multiple threads on uniprocessor systems might not explicitly protect the shared data. Instead, these applications might depend on the implicit protection provided by features of the VAX architecture that guarantee synchronization between application threads executing on a VAX uniprocessor system (described in Section 6.1.1).

6.1.1 VAX Architectural Features That Guarantee Atomicity

The following features of the VAX architecture provide synchronization among multiple threads of execution running on a uniprocessor system. (The VAX architecture does not extend this guarantee of atomicity to multiprocessor systems.)

- **Instruction atomicity**—Many of the instructions defined by the VAX architecture are capable of performing a read-modify-write operation in a single, noninterruptible sequence (called an **atomic** operation) from the

viewpoint of multiple application threads executing on a single processor. The Intel Itanium architecture supports such instructions.

To provide compatibility with VAX systems, the Intel Itanium architecture defines several instructions that you can use to ensure that a read/write operation is done atomically. Section 6.1.2 describes these instructions and how compilers on I64 systems make this capability available to programs written in high-level languages.

However, even on VAX systems, implicit dependence on the atomicity of VAX instructions is not recommended. Because of the optimizations they perform, compilers on VAX systems do not guarantee that they use a VAX atomic instruction to implement certain program statements, such as an increment operation (*x* = *x*+ 1), even if such an instruction is available.

- **Memory access granularity**—The VAX architecture supports instructions that can manipulate byte-sized and word-sized data in a single, noninterruptable operation. (The VAX architecture supports instructions to manipulate data of other sizes as well.) The Intel Itanium architecture also supports instructions that manipulate byte-sized and word-sized data.
- **Read/write ordering**—On VAX uniprocessor and multiprocessor systems, sequential write operations and read operations appear to occur in the same order in which you specify them from the viewpoint of all types of external threads of execution. I64 uniprocessor systems also guarantee that the order of read and write operations appears synchronized for multiple threads of execution running within a single process or within multiple processes running on a uniprocessor. However, write operations that are visible to threads executing concurrently on an I64 multiprocessor system require explicit synchronization.

  To provide compatibility with VAX systems, the Intel Itanium architecture supports an instruction that ensures that read/write operations occur in the order specified, from the viewpoint of all the processors in the system. Section 6.1.2 provides more information about this instruction and about how high-level languages make this instruction available. Section 6.3 describes the feature of the Intel Itanium architecture that provides this synchronization and how the compilers make it available to high-level language programs on I64 systems.

6.1.2 Intel Itanium Compatibility Features

The Intel Itanium architecture has several mechanisms to provide compatibility with the atomicity capabilities of the VAX architecture:

- **Compare and Exchange instructions**—The Intel Itanium instruction set defines four instructions, named Compare and Exchange (cmpxchg1, cmpxchg2, cmpxchg4, cmpxchg8) that provide for atomic compare and memory exchange operations.
- **Exchange instructions**—The Intel Itanium instruction set defines four instructions, named Exchange (xchg1, xchg2, xchg4, xchg8) that provide for atomic memory exchange operations.
- **Fetchadd instructions**—The Intel Itanium instruction set defines two instructions, named Fetch and Add Immediate (fetchadd4, fetchadd8) that provide for atomic increment or decrement of a memory location.

- **Memory fence**—The I64 instruction set includes an instruction that can ensure that read/write operations, issued by multiple threads executing on separate processors in a multiprocessor system, appear to occur in the order specified. This instruction, named Memory Fence (MF), guarantees that all subsequent load or store instructions do not access memory until after all previous load and store instructions have accessed memory from the viewpoint of multiple threads of execution.

  In addition to the MF instruction, all of the previously listed instructions have forms that ensure that the memory read/write is made visible either before all subsequent data memory accesses or after all previous data memory accesses.

6.2 Uncovering Atomicity Assumptions in Your Application

One way to uncover synchronization assumptions in your application is to identify data that is shared among multiple threads of execution and then examine each access to the data from each thread. When looking for shared data, remember to include unintentionally shared data as well as intentionally shared data. Shared data can be shared unintentionally because of its proximity to data that is accessed by multiple threads of execution, such as data written to by ASTs generated by the operating system as a result of system services such as $QIO, $ENQ, or $GETJPI.

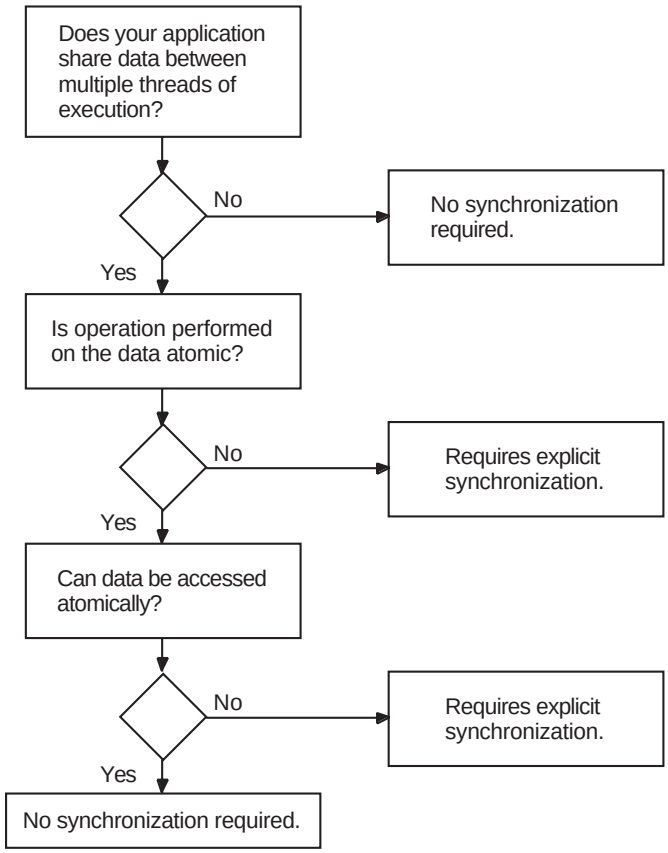
Because compilers on I64 systems use quadword instructions by default in certain circumstances, all data items within a quadword of a shared data item might become shared unintentionally. (For example, compilers use quadword instructions to access a data item that is not aligned on natural boundaries. Data is naturally aligned when its address is divisible by its size. For more information, see Chapter 7. Compilers align explicitly declared data on natural boundaries by default.)

When examining data access, determine whether another thread can view the data in an intermediate state and, if so, whether it is important to the application. In some cases, the exact value of the shared data might not be important; the application depends only on the relative value of the variable. In general, ask the following questions:

- Is the operation performed on the shared data atomic from the viewpoint of other threads of execution?
- Is it possible to perform an atomic operation to the data type involved?

Figure 6–1 shows this decision process.

Figure 6–1 Synchronization Decision Tree



ZK–5204A–GE

6.2.1 Protecting Explicitly Shared Data

Example 6–1 is a simplified example of some possible atomicity assumptions in a VAX application. The program uses a variable, called *flag*, through which an AST thread communicates with a main processing thread of execution. The main processing loop continues working until the counter variable reaches a predetermined value. The program queues an AST interruption that sets the flag to the maximum value, terminating the processing loop.

Example 6–1 Atomicity Assumptions in a Program with an AST Thread

```
#include <ssdef.h>
#include <descrip.h>

#define MAX_FLAG_VAL 1500

int    ast_rout();
long  time_val[2];
short int    flag;    /* accessed by main and AST threads */
```

(continued on next page)

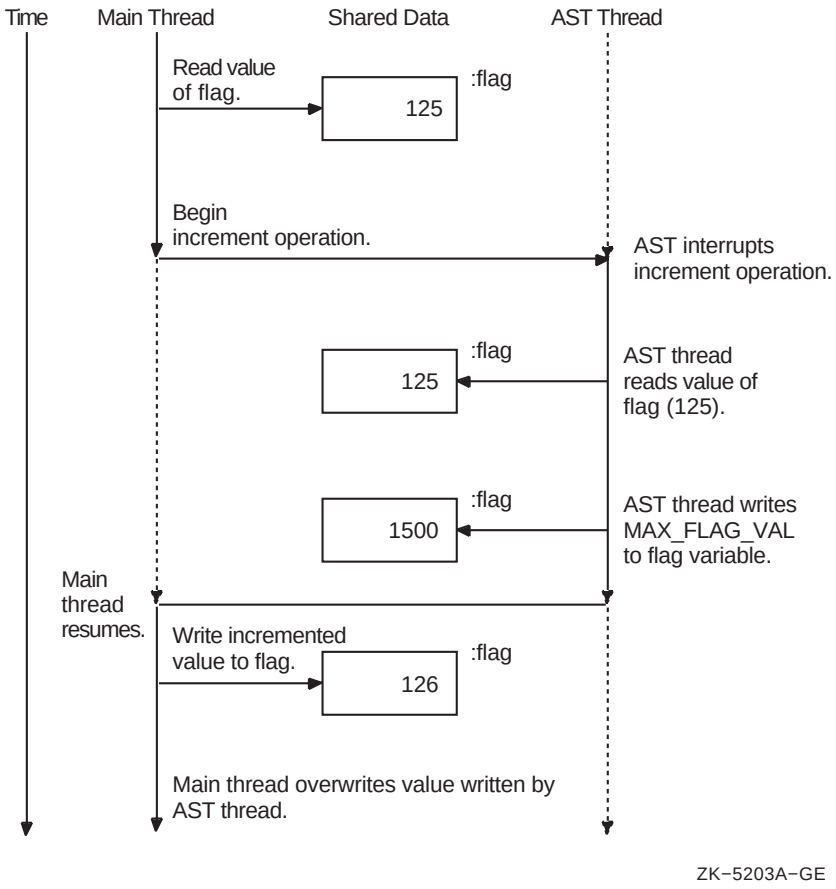
Example 6–1 (Cont.) Atomicity Assumptions in a Program with an AST Thread

```
main( )
{
     int      status = 0;
     static  $DESCRIPTOR(time_desc, "0 ::1");

     /*  changes ASCII time value to binary value  */

     status = SYS$BINTIM(&time_desc, &time_val);

     if ( status != SS$_NORMAL )
     {
        printf("bintim failure\n");
        exit( status );
     }

     /*  Set timer, queue ast */

     status = SYS$SETIMR( 0, &time_val, ast_rout, 0, 0 );

     if ( status != SS$_NORMAL )
     {
        printf("setimr failure\n");
        exit( status );
     }

     flag = 0;   /* loop until flag = MAX_FLAG_VAL */
     while( flag < MAX_FLAG_VAL )
     {
           printf("main thread processing (flag = %d)\n",flag);
           flag++;
     }
     printf("Done\n");
}

ast_rout()     /*  sets flag to maximum value to stop processing */
{
      flag = MAX_FLAG_VAL;
}
```

In Example 6–1, the *flag* variable is explicitly shared between the main thread of execution and an AST thread. The program does not use any synchronization mechanism to protect the integrity of this variable; it implicitly depends on the atomicity of the increment operation.

On I64 systems, this program might not always work as expected because the mainline thread of execution can be interrupted in the middle of the increment operation by the AST thread before the new value is stored back into memory, as shown in Figure 6–2. (This is more likely to fail in a real application with dozens of AST threads.) In this scenario, the AST thread interrupts the increment operation before it completes, setting the value of the variable to the maximum value. But once control returns to the main thread, the increment operation completes, overwriting the value of the AST thread. When the loop test is performed, the value is not at its maximum, and the processing loop continues.

Figure 6–2 Atomicity Assumptions in Example 6–1



Recommendations

To correct these atomicity dependencies, HP recommends doing the following:

- Disable AST delivery using the $SETAST system service, while the data is being accessed, and enable it after access is completed.
- Explicitly protect the data by using a compiler mechanism. For example, C for OpenVMS I64 systems supports atomicity built-ins. In addition, you can use other mechanisms to synchronize access to this data, such as the $ENQ system service (for data accessed by multiple threads running on a multiprocessor system) or run-time library routines, such as LIB$BBCCI or LIB$BBSSI, and the interlocked queue routines.

  For example, in Example 6–1, replace the increment operation, which is performed by the C increment operator (flag++), with the atomicity built-in supported by C for OpenVMS I64 systems (__ADD_ATOMIC_LONG(&flag,1,0)). See Example 6–2 for the complete example.

  Note that the shared variable must be an aligned longword or aligned quadword to be protected by the atomicity built-ins.

- If you cannot change byte-sized or word-sized data to a longword or quadword, then change the granularity the compiler uses when accessing the data item. Many compilers on I64 systems allow you to specify the granularity they use when accessing a particular data item or when processing an entire module. However, specifying byte and word granularity can have an adverse effect on the performance of your application.

Example 6–2 shows how these changes are implemented in the program presented in Example 6–1.

Example 6–2 Version of Example 6–1 with Synchronization Assumptions

```
#include <ssdef.h>
#include <descrip.h>
#include <builtins.h> ❶

#define MAX_FLAG_VAL  1500
int    ast_rout();
long  time_val[2];
int ❷       flag;    /* accessed by mainline and AST threads */

main( )
{
     int      status = 0;
     static  $DESCRIPTOR(time_desc, "0 ::1");

     /*  changes ASCII time value to binary value  */

     status = SYS$BINTIM(&time_desc, &time_val);

     if ( status != SS$_NORMAL )
     {
        printf("bintim failure\n");
        exit( status );
     }

     /*  Set timer, queue ast */

     status = SYS$SETIMR( 0, &time_val, ast_rout, 0, 0 );

     if ( status != SS$_NORMAL )
     {
        printf("setimr failure\n");
        exit( status );
     }

     flag = 0;
     while( flag < MAX_FLAG_VAL )   /* perform work until flag set to zero */
     {
           printf("mainline thread processing (flag = %d)\n",flag);
           __ADD_ATOMIC_LONG(&flag,1,0); ❸
     }
     printf("Done\n");
}

ast_rout()     /*  sets flag to maximum value to stop processing */
{
      flag = MAX_FLAG_VAL;
}
```

The following items correspond to Example 6–2:

❶ To use the C for OpenVMS I64 systems atomicity built-ins, you must include the builtins.h header file.

❷ In this version, the variable *flag* is declared as a longword to allow atomic access (the atomicity built-ins require it).

❸ The increment operation is performed with an atomicity built-in function.

6.2.2 Protecting Unintentionally Shared Data

In Example 6–1, both threads clearly access the same variable. However, on I64 systems, an application can have atomicity concerns for variables that are inadvertently shared. In this scenario, two variables are physically adjacent to each other within the boundaries of a longword or quadword. On VAX systems, each variable can be manipulated individually. On an I64 system, which supports atomic read and write operations of longword and quadword data only, the entire longword must be fetched before the target bytes can be modified. (For more information about this change in data-access granularity, see Chapter 7.)

To illustrate this problem, consider a modified version of the program in Example 6–1 in which the main thread and the AST thread each increment separate counter variables that are declared in a data structure, as in the following code:

```
struct {
    short int     flag;
    short int ast_flag;
    };
```

If both the main thread and the AST thread attempt to modify their individual target words simultaneously, the results are unpredictable, depending on the timing of the two operations.

Recommendations

To remedy this synchronization problem, HP suggests doing the following:

- Change the size of the shared variables to longwords or quadwords. However, because compilers on I64 systems use quadword instructions in certain circumstances, you should use quadwords to ensure the integrity of the data. For example, if the data is not aligned on a natural boundary, the compilers use a quadword instruction to access the data.

  In data structures, you can also insert extra bytes between data items to force the elements of the structure onto natural quadword boundaries. On I64 systems, the compilers align data on natural boundaries by default.

  For example, to ensure that each flag variable in the data structure can be modified without interference from other threads of execution, change the declarations of the variables so that they are 64-bit quantities. Using C, you could use the `double` data type, as in the following code:

  ```
  struct {
      double      flag;
      double  ast_flag;
      };
  ```

- Explicitly protect the data by using a compiler mechanism, such as the atomicity built-ins or the **volatile** attribute. In addition, you can synchronize access to data by multiple threads of execution running on a multiprocessor system by using the $ENQ system service or a run-time library routine, such as LIB$BBCCI or LIB$BBSSI, or by using interlocked queue operations.

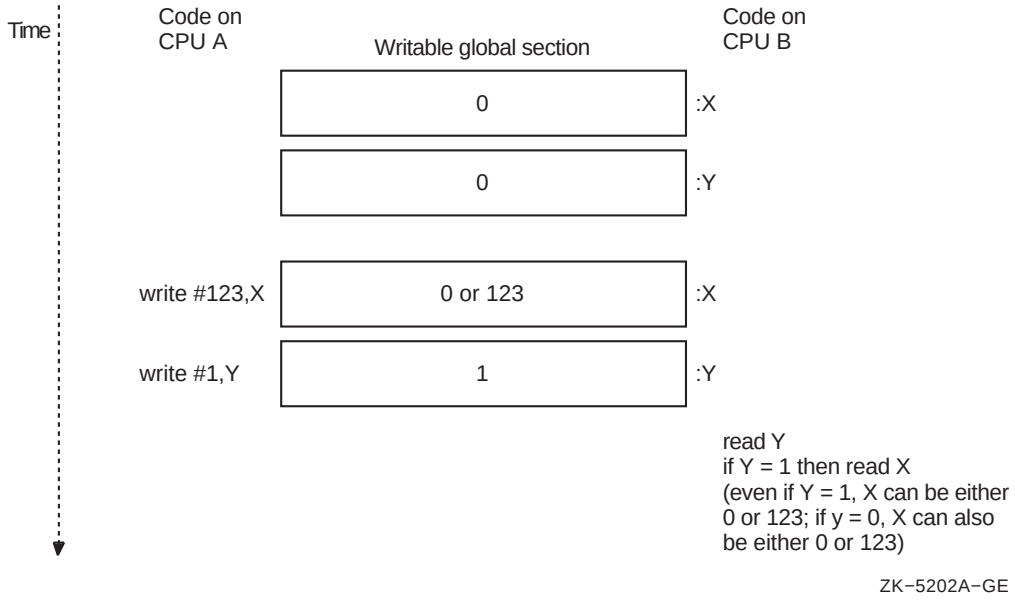
6.3 Synchronizing Read/Write Operations

VAX multiprocessing systems have traditionally been designed so that if one processor in a multiprocessing system writes multiple pieces of data, these pieces become visible to all other processors in the same order in which they were written. For example, if CPU A writes a data buffer (represented by X in Figure 6–3) and then writes a flag (represented by Y in Figure 6–3), CPU B can determine that the data buffer has changed by examining the value of the flag.

On I64 systems, read and write operations to memory can be reordered to benefit overall memory subsystem performance. Processes that execute on a single processor can rely on write operations from that processor becoming readable in the order in which they are issued. However, multiprocessor applications cannot rely on the order in which write operations to memory become visible throughout the system. In other words, write operations performed by CPU A might become visible to CPU B in an order different from that in which they were written.

Figure 6–3 depicts this problem. CPU A requests a write operation to X, followed by a write operation to Y. CPU B requests a read operation from Y and, seeing the new value of Y, initiates a read operation of X. If the new value of X has not yet reached memory, CPU B receives the old value. As a result, any token-passing protocol relied on by procedures running on CPUs A and B is broken. CPU A could write data and set a flag bit, but CPU B might see the flag bit set before the data is actually written and might erroneously use stale memory contents.

Figure 6–3 Order of Read and Write Operations on an I64 System



Recommendations

Programs that run in parallel and that rely on read/write ordering require some redesigning to run correctly on an I64 system. One or more of the following techniques might be appropriate, depending on the application:

- Use the I64 Memory Fence (MF) instruction before and after all read and write instructions for which the completion order is crucial. For example, the C for OpenVMS I64 systems compiler supports the MF instruction with the __MB() built-in.
- Redesign the application to use either built-in memory interlock operations provided by the compiler or the VAX interlocked instruction routines available in the LIB$ run-time library.
- Redesign the application to use the $ENQ and $DEQ system services to protect the data with a lock.

6.4 Ensuring Atomicity in Translated Images

The VEST command /PRESERVE qualifier accepts keywords that allow translated VAX images to run on Alpha systems with the same guarantees of atomicity that are provided on VAX systems. Several /PRESERVE qualifier keywords provide different types of atomicity protection. Note that specifying these /PRESERVE qualifier keywords can adversely affect the performance of your application. (For complete information about specifying the /PRESERVE qualifier, see *OpenVMS Migration Software for VAX to Alpha Systems: Translating Images*.)

To ensure that an operation that can be performed atomically on a VAX system by a VAX instruction is performed atomically in a translated image, specify the INSTRUCTION_ATOMICITY keyword to the /PRESERVE qualifier.

To ensure that simultaneous updates to adjacent bytes within a longword or quadword can be accomplished without interfering with each other, specify the MEMORY_ATOMICITY keyword to the /PRESERVE qualifier.

To ensure that read/write operations appear to occur in the order you specify them, specify the READ_WRITE_ORDERING keyword to the /PRESERVE qualifier.

7

Checking the Portability of Application Data Declarations

This chapter describes how to check the data your application uses for dependencies on the VAX architecture. The chapter also describes the effect your choice of data type can have on the size and performance of your application on an I64 system.

7.1 Overview

The data types supported by high-level programming languages, such as `int` in C or INTEGER*4 in Fortran, provide applications with a degree of data portability because they hide the machine-specific details of the underlying native data types. The languages map their data types to the native data types supported by the target platform. For this reason, you might be able to successfully recompile and run an application that runs on VAX systems on an I64 system without modifying the data declarations it contains.

However, if your application contains any of the following assumptions about data types, you might need to modify your source code:

- **Assumptions about data-type mappings**—Your application might depend on the underlying VAX data type to which a high-level language maps. The Intel Itanium architecture supports most of the VAX data types; however, some are not supported. Your application might make assumptions about the size or bit format of a data type that might no longer be valid on an I64 system. Section 7.2 provides more information about this topic.
- **Assumptions about data-type selection**—Your choice of data type might have different implications on an I64 system. For example, on VAX systems, you might have chosen the smallest data type available to represent data items to conserve memory usage. On an I64 system, this strategy may actually increase memory usage. Section 7.3 provides more information about this topic.

7.2 Checking for Dependence on a VAX Data Type

To provide data compatibility, the Intel Itanium architecture has been designed to support many of the same native data types as the VAX architecture. Table 7–1 lists the native data types supported by both architectures. (See Appendix B of the *HP OpenVMS Programming Concepts Manual* for more information about the formats of the data types.)

Table 7–1 Comparison of VAX and I64 Native Data Types

VAX Data Types	I64 Data Types
byte	byte
word	word
longword	longword
quadword	quadword
octaword	–
F_floating	F_floating [1]
D_floating (56-bit precision)	D_floating (56-bit precision)[1]
G_floating	G_floating [1]
H_floating	–
–	S_floating (IEEE)
–	T_floating (IEEE)
–	X_floating (IEEE)
Variable-length bit field	–
Absolute queue	Absolute longword queue
–	Absolute quadword queue
Self-relative queue	Self-relative longword queue
–	Self-relative quadword queue
Character string	–
Trailing numeric string	–
Leading separate numeric string	–
Packed decimal string	–

[1]Data types not supported by hardware. Support is provided by the compiler.

Recommendations

Unless your application depends on the format or size of the underlying native VAX data types, you might not have to modify your application because of changes to the data-type mappings. Wherever possible, the compilers on I64 systems map their data types to the same native data types as they do on VAX systems. For VAX data types that are not supported by the Intel Itanium architecture, the compilers map their data types to the closest equivalent native I64 data type. (For more information about how the I64 compilers systems map supported data types to native I64 data types, see Chapter 9 and compiler documentation.)

The following list provides guidelines that can be helpful for certain types of data declarations:

- **VAX floating-point data types**—See the "OpenVMS Floating-Point Arithmetic on the Intel® Itanium® Architecture" white paper for information about support for VAX floating-point data types on Integrity servers. See the Preface of this manual for the Web location of this white paper.)
- **Pointer data**—Check for assumptions that an address (pointer) data type is equivalent in size to an integer data type. On I64 systems, it is possible to use 64-bit addresses. For information about converting your applications to use 64-bit pointers, see to the *HP OpenVMS Programming Concepts Manual*.

7.3 Examining Assumptions about Data-Type Selection

Even though your application might recompile and run successfully on an I64 system, your data-type selection might not take full advantage of the benefits of the Intel Itanium architecture. In particular, data-type selection can impact the ultimate size of your application and its performance on I64 systems.

7.3.1 Effect of Data-Type Selection on Code Size

On VAX systems, applications typically use the smallest-size data type adequate for the data. For example, to represent a value between 32,768 and -32,767 in an application written in C, you might declare a variable of type short. On VAX systems, this practice conserves storage and, because the VAX architecture supports instructions that operate on all sizes of data types, does not affect efficiency.

You do not need to promote byte and word fields to longword or quadword fields because I64 also supports byte and word instructions. One reason requiring field promotion is to be able to fit larger values, but it is not necessary to promote all byte and word fields.

7.3.2 Effect of Data-Type Selection on Performance

Another aspect of data-type selection is data alignment. Alignment is an attribute of a data item that refers to its placement in memory. The mixture of byte-sized, word-sized, and larger data types, typically found in data-structure definitions and static data areas in applications on VAX systems, can lead to data that is not aligned on natural boundaries. (A data item is naturally aligned when its address is a multiple of its size in bytes.)

On both VAX and I64 systems, accessing unaligned data incurs more overhead than accessing aligned data . However, VAX systems use microcode to handle and fixup unaligned data. On I64 systems, hardware assistance is not available. References to unaligned data trigger a fault, which must be handled by the operating system. Thus, the cost of an unaligned reference in performance is dramatically higher on I64 systems.

The compilers on I64 systems attempt to minimize the performance impact by generating a special unaligned reference instruction sequence when an unaligned reference is known at compile time. This prevents the occurrence of a run-time unaligned fault. Unaligned references that appear at run time must be handled as unaligned reference faults.

Recommendations

Given the potential impact of data-type selection on code size and performance, you might think you should change all byte-sized and word-sized data declarations to longwords to eliminate the extra instructions required for byte and word accesses and improve alignment. However, before making sweeping changes to your data declarations, consider the following factors:

- **Frequency of access/number of replications**—If a byte-sized or word-sized data item is referenced frequently, changing it to a longword eliminates the requirement for extra instructions at each reference and can reduce application size significantly. However, if the byte or word is not referenced frequently and is replicated a large number of times (for example, in a data structure that is instantiated many times), the change to a longword can add up to more than the cost of the additional instructions at each reference. The three bytes added when changing to a longword can significantly increase virtual memory usage if the data item is replicated thousands of times. Before

changing a data declaration, consider how it is used and how much virtual memory (and thus physical memory) you want to spend for this performance improvement. Such trade-offs between size and performance are a frequent consideration during design.

- **Interoperability requirements**—If the data object is shared with a translated component or a native VAX component, you might be unable to make changes that would improve its layout because the other components depend on the binary layout of the data. Compilers (and the VEST utility) attempt to minimize the performance impact in this case by including the unaligned reference instruction sequence in the code they generate.

Taking these factors into consideration, use the following guidelines when examining data-type selections:

- For data that is frequently referenced but not frequently replicated, change byte-sized and word-sized fields to longwords, especially for performance-critical fields.
- For data that is not frequently referenced but that is frequently replicated, no change is recommended.
- For data that is both frequently referenced and frequently replicated, the decision must be made after carefully examining the code size versus performance impact of the change.
- Use the capabilities of the compilers on I64 systems to uncover data that is not aligned on natural boundaries. Many compilers on I64 systems (for example, HP Fortran) support the /WARNING=ALIGNMENT qualifier, which checks for data that is not aligned on natural boundaries).
- Use the capabilities of the run-time analysis tools, Program Coverage and Analyzer (PCA) and the OpenVMS Debugger, to uncover at run time data that is not aligned on natural boundaries. For more information, see the *Guide to Performance and Coverage Analyzer for VMS Systems* and the *HP OpenVMS Debugger Manual*.
- Take advantage of the natural alignment provided by the compilers on I64 systems, wherever interoperability concerns allow. On I64 systems, compilers align data on natural boundaries by default, wherever possible. On VAX systems, compilers use byte alignment.

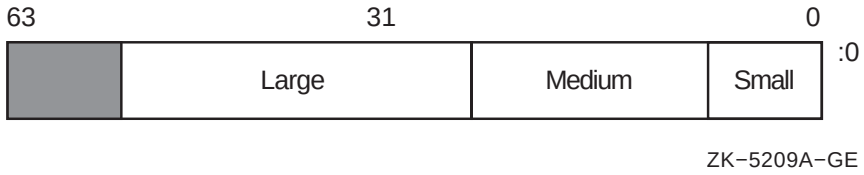
  Compilers on I64 systems support qualifiers and language pragmas that allow you to request they use the same byte alignment they use on VAX systems. For example, the C compiler for OpenVMS I64 systems supports the /NOMEMBER_ALIGNMENT qualifier and a corresponding pragma that allow you to control data alignment. For more information, see the C compiler documentation.

The following example data structure, called mystruct, is made up of byte-sized, word-sized, and longword-sized data:

```
struct{
      char     small;
      short   medium;
      long     large;
      } mystruct ;
```

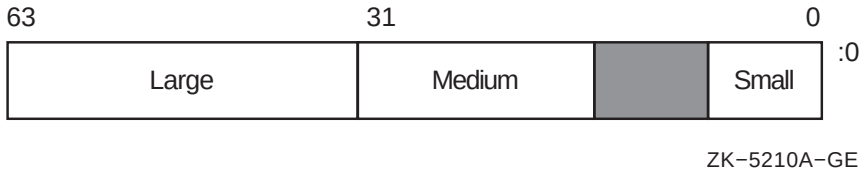
Figure 7–1 shows how the structure is laid out in memory when it is compiled using VAX C.

Figure 7–1 Alignment of mystruct Using VAX C



When compiled using the C compiler for OpenVMS I64 systems, this structure is padded to achieve natural alignment, as shown in Figure 7–2. Note that by adding a byte of padding after the first field (Small), both the next two fields of the structure are aligned.

Figure 7–2 Alignment of mystruct Using C for OpenVMS I64 Systems



Note that the byte-sized and word-sized fields of the data structure still require multiple instruction sequences for access. If the fields Small and Medium are frequently referenced, and the entire structure is not frequently replicated, consider redefining the data structure to use longword data types. However, if the fields are not frequently referenced or the data structure is frequently replicated, the cost of the byte or word references is a design trade-off the programmer must make.

8

Examining the Condition-Handling Code in Your Application

This chapter describes the effect of differences between the VAX architecture and the Intel Itanium architecture on the condition-handling code in your application.

8.1 Overview

For the most part, the condition-handling code in your application will work correctly on I64 systems, especially if your application uses the condition-handling facilities provided by the high-level language in which it is written, such as the END, ERR, and IOSTAT specifiers in Fortran. These language capabilities insulate applications from architecture-specific aspects of the underlying condition-handling facility.

However, there are certain differences between the I64 condition-handling facility and the VAX condition-handling facility that might require you to modify your source code. Such changes include:

- Changes to the mechanism array format
- Changes to the condition codes returned by the system
- Changes to how other tasks related to condition handling in your application are accomplished, such as enabling exception signaling and specifying condition-handling routines dynamically at run time

The following sections describe these changes and provide guidelines to help you decide whether modifications are necessary.

8.2 Establishing Dynamic Condition Handlers

The OpenVMS I64 run-time libraries (RTLs) do not contain the routine LIB$ESTABLISH, which the OpenVMS VAX RTLs contain. Because of the nature of the OpenVMS I64 calling standard, setting up condition handlers is done by compilers.

For programs that need to dynamically establish condition handlers, some I64 languages give special treatment for calls to LIB$ESTABLISH and generate the appropriate code without actually calling an RTL routine. The following languages support LIB$ESTABLISH semantics in a compatible fashion with the corresponding VAX language:

- HP C and HP C++

  Although HP C and HP C++ for OpenVMS I64 systems treat LIB$ESTABLISH as a built-in function, the use of LIB$ESTABLISH is not recommended on OpenVMS VAX or OpenVMS I64 systems. C and C++ programmers should call VAXC$ESTABLISH instead of LIB$ESTABLISH

(VAXC$ESTABLISH is a built-in function on HP C and HP C++ for OpenVMS I64 systems).

- HP Fortran

  HP Fortran allows declarations to the LIB$ESTABLISH and LIB$REVERT intrinsic functions, and converts them to HP Fortran RTL-specific entry points.

- HP Pascal

  HP Pascal provides the built-in routines, ESTABLISH and REVERT, to use in place of LIB$ESTABLISH and LIB$REVERT. If you declare and try to use LIB$ESTABLISH, you get a compile-time warning.

- MACRO–32

  The MACRO–32 compiler attempts to call LIB$ESTABLISH if it is contained in the source code.

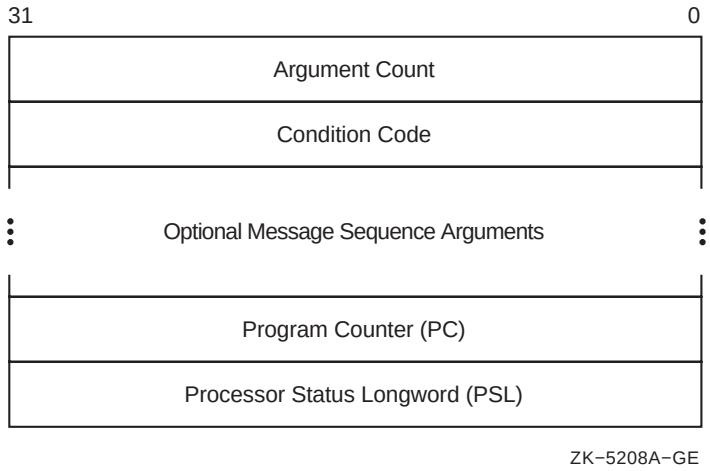
  If MACRO–32 programs establish dynamic handlers by storing a routine address at 0(FP), they work correctly when compiled on an OpenVMS I64 system. However, you cannot set the condition handler address from within a JSB (Jump to Subroutine) routine, only from within a CALL_ENTRY routine.

8.3 Examining Condition-Handling Routines for Dependencies

The calling sequence of user-written condition-handling routines remains the same on I64 systems as on VAX systems. Condition-handling routines declare two arguments to access the data the system returns when it signals an exception condition. The system uses two arrays, the signal array and the mechanism array, to convey information that identifies which exception condition triggered the signal and to report on the state of the processor when the exception occurred.

The format of the signal array and the mechanism array is defined by the system and is documented in the *HP OpenVMS Programming Concepts Manual*. On I64 systems, the data returned in the signal array and its format is the same as it is on VAX systems, as shown in Figure 8–1.

Figure 8–1 32-Bit Signal Array on VAX and I64 Systems



The following table describes the arguments in the signal array:

Argument	Description
Argument Count	On I64 and VAX systems, this argument contains a positive integer that indicates the number of longwords that follow in the array.
Condition Code	On I64 and VAX systems, this argument is a 32-bit code that uniquely identifies a hardware or software exception condition. The format of the condition code, which remains unchanged on I64 systems, is described in *OpenVMS Programming Interfaces: Calling a System Routine*. Note that I64 systems do not support every condition code returned on VAX systems and define condition codes that cannot be returned on a VAX system. Section 8.4 lists VAX condition codes that cannot be returned on I64 systems.
Optional Message Sequence	These arguments provide additional information about the particular exception returned and vary for each exception. The *HP OpenVMS Programming Concepts Manual* describes these arguments.
Program Counter (PC)	The address of the next instruction to be executed when the exception occurred, if the exception is a trap; or the address of the instruction that caused the exception, if the exception is a fault. On I64 systems, this argument contains the lower 32 bits of the PC (which is 64 bits long on I64 systems).
Processor Status Longword (PSL)	A formatted 32-bit argument that describes the status of the processor when the exception occurred. On I64 systems, this argument contains the lower 32 bits of an Alpha-equivalent processor status (PS) quadword. The IPL, CM, CSW, and IP fields are valid.

On I64 systems, the mechanism array returns much of the same data that it does on VAX systems; however, its format is different. The mechanism array returned on I64 systems preserves the contents of a larger set of integer scratch registers as well as the floating-point scratch registers. In addition, because these registers are 64 bits long, the mechanism array is constructed of quadwords (64 bits) on I64 systems, not longwords (32 bits) as it is on VAX systems. Figure 8–2 shows the format of the mechanism array on VAX systems. Figure 8–3 shows the format of the mechanism array on I64 systems.

Figure 8–2 Mechanism Array on VAX Systems

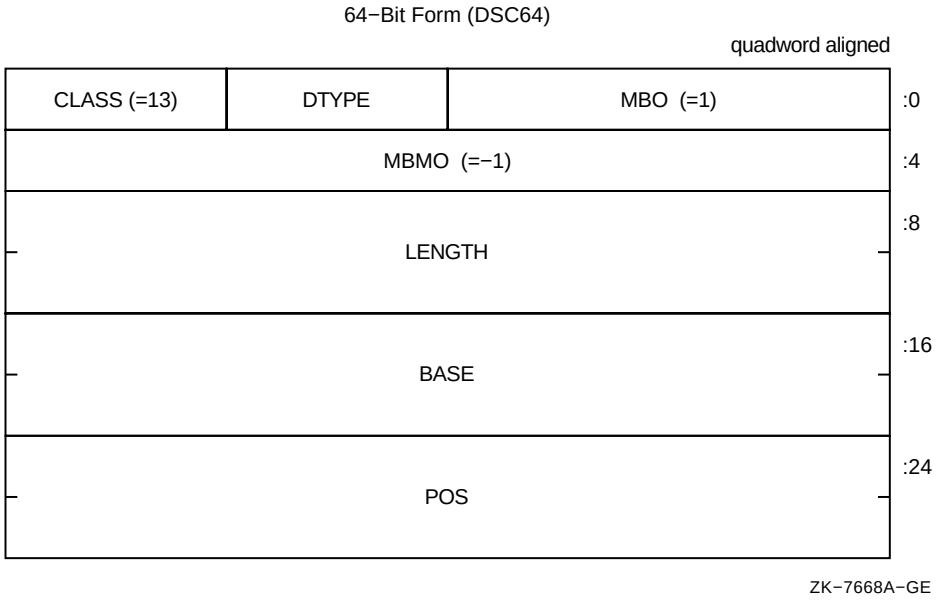
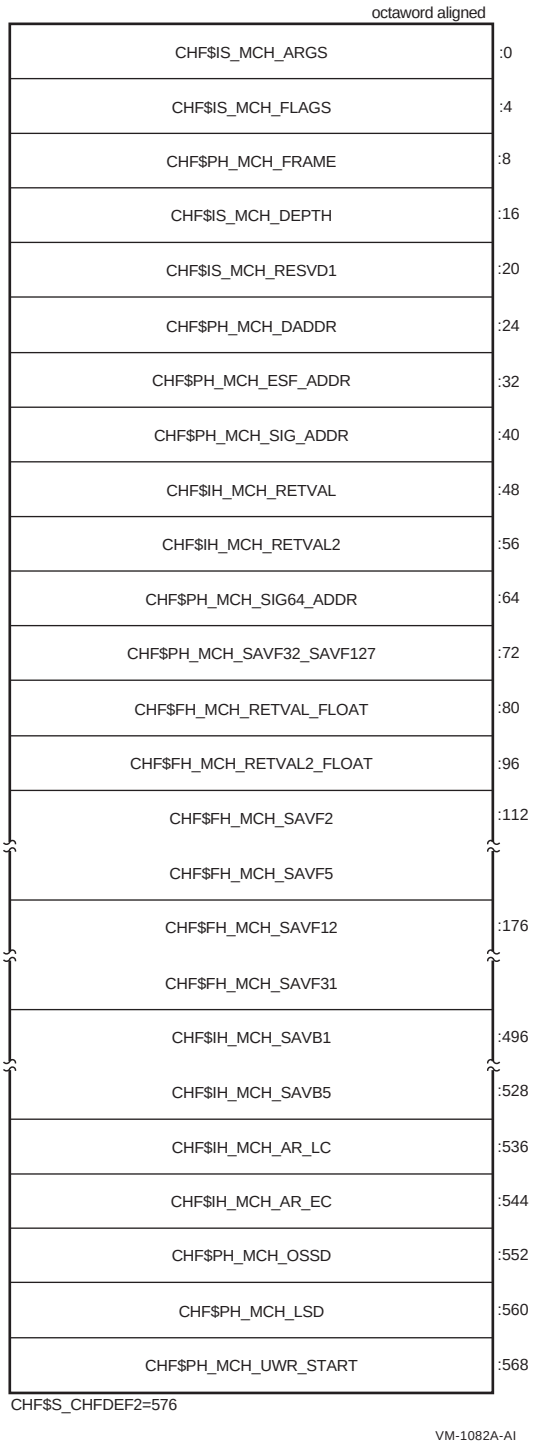


Figure 8–3 Mechanism Array on I64 Systems



The following table describes the arguments in the mechanism array:

Argument	Description
Argument Count	On VAX systems, contains a positive integer that represents the number of longwords that follow in the array. On I64 systems, represents the number of *quadwords* in the mechanism array, not counting the argument count quadword. On I64 systems, the argument count is 71 if CHF$V_FPREGS_VALID is clear and 263 if that same bit is set.
Flags	On I64 systems, contains various flags to communicate additional information. Bit 0 indicates that the process has already performed a floating-point operation in registers F2-F31 and the corresponding floating-point registers in the array are valid (no equivalent in the mechanism array on VAX systems). Bit 1 indicates that the process has already performed a floating-point operation in registers F32-F127 and the corresponding floating-point registers in the extension area are valid (no equivalent in the mechanism array on VAX systems).
Frame Pointer (FP)	On VAX, contains the contents of the FP. On I64, contains the Previous Stack Pointer (PSP), which is the value of the SP at procedure entry.
Depth	On VAX and I64 systems, contains an integer that represents the frame number of the procedure that established the condition-handling routine, relative to the frame that incurred the exception.
Reserved	Reserved.
Handler Data Address	On I64 systems, contains the address of the handler data quadword, when a handler is present (no equivalent in the mechanism array on VAX systems).
Exception Stack Frame Address	On I64 systems, contains the address of the exception stack frame (no equivalent in the mechanism array on VAX systems).
Signal Array Address	On I64 systems, contains the address of the 32-bit signal array (no equivalent in the mechanism array on VAX systems).
Function Return Value	On I64 systems, these two quadwords contain the contents of R8 and R9 at the time of the exception (no equivalent in the mechanism array on VAX systems).
Signal Array Address	On I64 systems, contains the address of the 64-bit signal array (no equivalent in the mechanism array on VAX systems).
Floating Point Extension Address	On I64 systems, contains the address of the array containing the contents of floating point registers F32-F127 at the time of the exception (no equivalent in the mechanism array on VAX systems).
Floating Return Value	On I64 systems, these two quadwords contain the contents of F8 and F9 at the time of the exception (no equivalent in the mechanism array on VAX systems).
Registers	On VAX and I64 systems, the mechanism array includes the contents of scratch registers. On I64 systems, includes a much larger set of registers, as well as floating-point registers, branch registers, and some application registers.

Argument	Description
Operating System-specific Data Area	On I64 systems, contains the address of the operating system-specific data area of the condition handler (no equivalent in the mechanism array on VAX systems).
Invocation Handle	On I64 systems, contains the invocation handle of the procedure that established the condition handler (no equivalent in the mechanism array on VAX systems).
Unwind Region	Address of the unwind region.

For complete information, see the *HP OpenVMS Calling Standard*.

Recommendations

Because the 32-bit signal array is the same on I64 systems as it is on VAX systems, you might not need to modify the source code of your condition-handling routine. However, changes to the mechanism array might require changes to your source code. In particular, make note of the following:

- Check the source code of your condition-handling routine for assumptions about the size of array elements or the ordering of array elements in the mechanism array.
- If the condition-handling routine in your application uses the depth argument to unwind a specific number of stack frames, you might need to modify your source code. Because of architectural changes, the depth argument returned on I64 systems might be different from that returned on VAX systems. (The depth argument in the mechanism array indicates the number of frames between the procedure that established the handler, relative to the frame that incurred the exception.)

  Applications that unwind to the establisher frame by specifying the address of the depth argument to the SYS$UNWIND system service, or that unwind to the caller of the establisher frame by using the default depth argument of the SYS$UNWIND system service, continue to work correctly. Depths specified as negative numbers still indicate exception vectors (as on VAX systems).

Example 8–1 presents a condition-handling routine written in C.

Example 8–1 Condition-Handling Routine in C

```
#include  <ssdef.h>
#include  <chfdef.h>
   .
   .
   .
❶ int cond_handler( sigs, mechs )
   struct  chf$signal_array  *sigs;
   struct  chf$mech_array   *mechs;
{
     int status;

❷    status = LIB$MATCH_COND(sigs->chf$l_sig_name, /* returned code */
                             SS$_INTOVF);          /* test against  */

❸    if(status != 0)
     {
         /*  ...Condition matched.  Perform processing.  */
         return SS$_CONTINUE;
     }
     else
     {
         /*  ...Condition does not match. Resignal exception. */
         return SS$_RESIGNAL;
     }
}
```

The following items correspond to Example 8–1:

❶ The routine defines two arguments, **sigs** and **mechs**, to access the data returned by the system in the signal array and the mechanism array. The routine declares the arguments using two predefined data structures, chf$signal_array and chf$mech_array, defined by the system in the CHFDEF.H header file.

❷ This condition-handling routine uses the LIB$MATCH_COND run-time library routine to compare the returned condition code with the condition code that identifies integer overflow (defined in SSDEF.H). The condition code is referenced as a field in the system-defined signal data structure (defined in CHFDEF.H).

❸ The LIB$MATCH_COND routine returns a nonzero result when a match is found. The condition-handling routine executes different code paths based on this result.

8.4 Identifying Exception Conditions

Application condition-handling routines identify which exception is being signaled by checking the condition code returned in the signal array. The following program fragment, taken from Example 8–1, shows how a condition-handling routine can accomplish this task by using the LIB$MATCH_COND run-time library routine.

```
status = LIB$MATCH_COND( sigs->chf$l_sig_name,  /* returned code */
                                 SS$_INTOVF);  /* test against  */
```

On I64 systems, the format of the 32-bit condition code and its location in the signal array are the same as on VAX systems. However, the condition codes your condition-handling routine expects to receive on VAX systems might not be meaningful on I64 systems. Because of architectural differences, some exception conditions that are returned on VAX systems are not supported on I64 systems.

For software exceptions, I64 systems support the same set supported by VAX systems, as documented in the online Help Message utility or in the OpenVMS system messages documentation. Hardware exceptions are more architecture specific, especially the arithmetic exceptions. Only a subset of the hardware exceptions supported by VAX systems (documented in the *HP OpenVMS Programming Concepts Manual*) are also supported on I64 systems. In addition, the Intel Itanium architecture defines several additional exceptions that are not supported by the VAX architecture.

Table 8–1 lists the VAX hardware exceptions that are not supported on I64 systems and the I64 hardware exceptions that are not supported on VAX systems. If the condition-handling routine in your application tests for any of these VAX-specific exceptions, you might need to add the code to test for the equivalent I64 exceptions. (Section 8.4.1 provides more information about testing for arithmetic exceptions on I64 systems.)

Note

A translated VAX image run on an I64 system can still return these VAX exceptions.

Table 8–1 Architecture-Specific Hardware Exceptions

Exception Condition Code	Comment
Exceptions Specific to I64 Systems	
SS$_FLTINV–Invalid floating-point operand value (trap)	No equivalent on VAX systems
SS$_FLTINV_F–Invalid floating-point operand value (fault)	No equivalent on VAX systems
SS$_FLTINE–Inexact floating-point result (trap)	No equivalent on VAX systems
SS$_FLTINE_F–Inexact floating-point result (fault)	No equivalent on VAX systems
SS$_FLTDENORMAL–Unnormalized floating-point result	No equivalent on VAX systems
SS$_ALIGN–Data-alignment trap	No equivalent on VAX systems
Exceptions Specific to VAX Systems	
SS$_ARTRES–Reserved arithmetic trap	No equivalent on I64 systems
SS$_COMPAT–Compatibility fault	No equivalent on I64 systems
SS$_DECOVF–Decimal overflow[1]	Not generated by I64 hardware
SS$_INTDIV–Integer divide-by-zero[1]	Not generated by I64 hardware
SS$_INTOVF–Integer overflow[1]	Not generated by I64 hardware
SS$_TBIT–Trace pending	No equivalent on I64 systems
SS$_OPCCUS–Opcode reserved to customer	No equivalent on I64 systems

[1]Might be generated by software on I64 systems

(continued on next page)

Table 8–1 (Cont.) Architecture-Specific Hardware Exceptions

Exception Condition Code	Comment
Exceptions Specific to VAX Systems	
SS$_RADMOD–Reserved addressing mode	No equivalent on I64 systems
SS$_SUBRNG–INDEX subscript range check	No equivalent on I64 systems

8.4.1 Testing for Arithmetic Exceptions on I64 Systems

While OpenVMS I64 uses most of the same floating point condition codes as OpenVMS VAX, it uses some new ones as well. In addition, in some cases conditions might be signalled by library software where they were caused by hardware on the VAX; the same circumstances might result in different exceptions on the two architectures.

Arithmetic exceptions on the Intel Itanium architecture are precise, as they are on the VAX; that is, the exception PC provides an exact indication of the failing instruction. (This is in contrast to the Alpha architecture, where floating point exceptions are asynchronous and imprecise.) However, because of the I64 instruction format, the exception PC is not simply the address of the failing instruction. I64 instructions are organized into 16 byte bundles of 3 instructions each. The exception PC is the address of the instruction bundle, with the instruction slot number in the low 2 bits.

Because of the differences in instruction addressing and instruction format, any condition handler code that attempts to interpret the instruction causing the exception must be rewritten.

Recommendations

The following guidelines can help you determine whether a condition-handling routine that performs processing in response to an arithmetic exception needs modification to run on I64 systems:

- If the condition-handling routine in your application only counts the number of arithmetic exceptions that occurred, or aborts when an arithmetic exception occurs, it requires minimal modification to work correctly on I64 systems. These condition-handling routines require only the addition of a test for the new I64 condition codes.
- If your application attempts to restart the operation that caused the exception, you must rewrite any portion of the condition handler that attempts to locate or interpret the failing instruction.

Note

A translated VAX image running on an I64 system can return VAX exception conditions, including arithmetic exception conditions. For more information about using the VEST command, see *OpenVMS Migration Software for VAX to Alpha Systems: Translating Images*.

8.4.2 Testing for Data-Alignment Traps

On I64 systems, a data-alignment trap is generated when an attempt is made to load or store a longword or quadword to or from a register using an address that does not have the natural alignment of the particular data reference. (For more information about data alignment, see Chapter 7.)

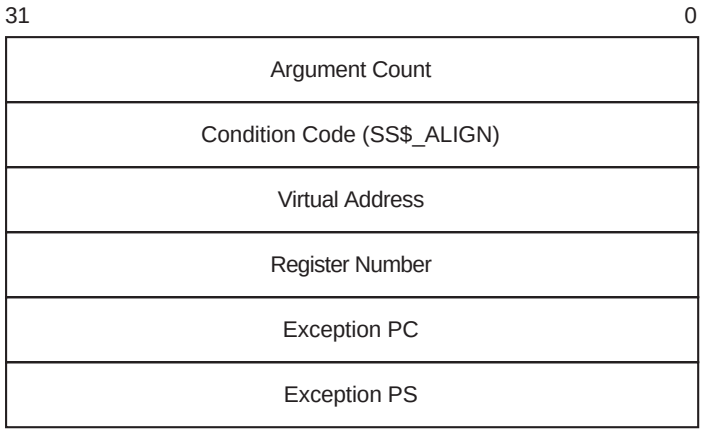
Compilers on I64 systems typically avoid triggering alignment faults by doing the following:

- Aligning static data on natural boundaries by default. (This default behavior can be overridden by using a compiler qualifier.)
- Generating special inline code sequences for data that is known to be misaligned at compile time.

Note that compilers cannot align dynamically defined data. Thus, alignment faults might be triggered.

An alignment exception is identified by the condition code SS$_ALIGN. Figure 8–4 shows the elements of the signal array returned by the SS$_ALIGN exception.

Figure 8–4 SS$_ALIGN Exception Signal Array



This signal array contains two arguments specific to the SS$_ALIGN exception: the **virtual address** argument and the **register number** argument. The virtual address argument contains the address of the unaligned data being accessed. The **register number** argument identifies the target register of the operation.

Recommendation

- Use this exception to detect alignment exceptions during the development of your application. In this phase, you have the opportunity to fix the data alignment before it impacts performance for a user of your application. Once this exception is reported, your application has already experienced the performance impact.

8.5 Performing Other Tasks Associated with Condition Handling

In addition to condition-handling routines, applications that include condition handling must perform other tasks, such as identifying their condition-handling routine to the system. The run-time library provides a set of routines that allows applications to perform these tasks. For example, applications can call the run-time library routine LIB$ESTABLISH to identify (or establish) the condition-handling routine they want executed when an exception is signaled.

Because of differences between the VAX architecture and the Intel Itanium architecture and between the calling standards for both architectures, the way in which many of these tasks are accomplished is not the same. Table 8–2 lists the run-time library condition-handling support routines available on VAX systems and indicates which are supported on I64 systems.

Table 8–2 Run-Time Library Condition-Handling Support Routines

Routine	Support on I64 Systems
Arithmetic Exception Support Routines	
LIB$DEC_OVER–Enable or disable signaling of decimal overflow	Not supported.
LIB$FIXUP_FLT–Change floating-point reserved operand to a specified value	Not supported.
LIB$FLT_UNDER–Enable or disable signaling of floating-point underflow	Not supported.
LIB$INT_OVER–Enable or disable signaling of integer overflow	Not supported.
General Condition-Handling Support Routines	
LIB$DECODE_FAULT–Analyze instruction context for fault	Not supported.
LIB$ESTABLISH–Establish a condition handler	Not supported by RTL but supported by compilers to provide compatibility.
LIB$MATCH_COND–Match condition value	Supported.
LIB$REVERT–Delete a condition handler	Not supported by RTL but supported by compilers to provide compatibility.
LIB$SIG_TO_STOP–Convert a signaled condition to a condition that cannot be continued	Supported.
LIB$SIG_TO_RET–Convert a signal to a return status	Supported.
LIB$SIM_TRAP–Simulate a floating-point trap	Not supported.
LIB$SIGNAL–Signal an exception condition	Supported.
LIB$STOP–Stop execution by using signaling	Supported.

Recommendations

The following guidelines apply to applications that use run-time library routines:

- If your application enables the signaling of exceptions by calling one of the run-time library routines that enable exception reporting, you must change your source code. These routines are not supported on I64 systems. Note, however, that certain types of arithmetic exceptions are always enabled on

I64 systems. The following types of arithmetic exceptions are always enabled:

- Floating-point invalid operation
- Floating-point division by zero
- Floating-point overflow

Exceptions that are not enabled by default must be enabled at compile time.

- If your application specifies a condition-handling routine by calling the LIB$ESTABLISH run-time library routine, you might not have to change your source code. To preserve compatibility, most compilers on I64 systems accept calls to the LIB$ESTABLISH routine. The compilers create a variable on the stack to point at the "current" condition handler. LIB$ESTABLISH sets this variable; LIB$REVERT clears it. The statically established handler for these languages reads the value of this variable to determine which routine to call. For information about specific languages, see Chapter 9.

The Fortran program in Example 8–2 uses the RTL routine LIB$ESTABLISH to specify a condition-handling routine that tests for integer overflow by specifying the condition code SS$_INTOVF. On VAX systems, you must compile the program with the /CHECK=OVERFLOW qualifier to enable integer overflow detection.

For this program to run on I64 systems, as VAX systems, you must specify the /CHECK=OVERFLOW qualifier on the compile command line to enable overflow detection. The call to the LIB$ESTABLISH routine does not have to be removed because HP Fortran accepts this routine as an intrinsic function.

Example 8–2 Sample Condition-Handling Program

```
C       This program types a maximum value of integers
C       Compile with /CHECK=OVERFLOW and the /EXTEND_SOURCE qualifiers

        INTEGER*4 int4
        EXTERNAL HANDLER
        CALL LIB$ESTABLISH (HANDLER)  ❶

        int4=2147483645
        WRITE (6,*) ' Beginning DO LOOP, adding 1 to ', int4
        DO I=1,10
          int4=int4+1
          WRITE (6,*) ' INT*4 NUMBER IS  ', int4
        END DO
        WRITE (6,*) ' The end ...'
        END

C       This is the condition-handling routine

        INTEGER*4 FUNCTION HANDLER (SIGARGS, MECHARGS)
        INTEGER*4 SIGARGS(*),MECHARGS(*)
        INCLUDE '($FORDEF)'
        INCLUDE '($SSDEF)'
        INTEGER INDEX
        INTEGER LIB$MATCH_COND
```

(continued on next page)

Example 8–2 (Cont.) Sample Condition-Handling Program

```
        INDEX = LIB$MATCH_COND (SIGARGS(2), SS$_INTOVF)
        IF (INDEX .EQ. 0 ) THEN
           HANDLER = SS$_RESIGNAL
        ELSE IF (INDEX .GT. 0) THEN
           WRITE (6,*) 'Arithmetic exception detected...'
           CALL LIB$STOP(SIGARGS(1))
        END IF
        END
```

The following item corresponds to Example 8–2:

❶ The example calls LIB$ESTABLISH to specify the condition-handling routine.

The following example shows how to compile, link, and run the program in Example 8–2.

```
$ FORTRAN/EXTEND_SOURCE/CHECK=OVERFLOW  EXCEPTIION
$ LINK  EXCEPTION
$ RUN EXCEPTION
 Beginning DO LOOP, adding 1 to 2147483645
 INT*4 NUMBER IS    2147483646
 INT*4 NUMBER IS    2147483647
Arithmetic exception detected...
%TRACE-F-TRACEBACK, symbolic stack dump follows
image     module    routine              line      rel PC           abs PC
exception  exception$MAIN  HANDLER       2662 0000000000000440 0000000000010440
DEC$FORRTL                                  0 00000000000DAC00 FFFFFFFF85440C00
                                            0 FFFFFFFF8039FE70 FFFFFFFF8039FE70
image     module    routine              line      rel PC           abs PC
exception  exception$MAIN  EXCEPTION$MAIN
                                           10 00000000000001C0 00000000000101C0
                                            0 FFFFFFFF80B356B0 FFFFFFFF80B356B0
DCL                                         0 000000000006AE90 000000007AE26E90
%TRACE-I-END, end of TRACE stack dump
```

9
OpenVMS I64 Compilers

This chapter provides information about features that are specific to native OpenVMS I64 compilers. In addition, it lists the features of OpenVMS VAX compilers that are not supported by or that have changed behavior in their OpenVMS I64 counterparts.

The following compilers are covered in this chapter:

- Ada (Section 9.1)
- BASIC (Section 9.2)
- C (Section 9.3)
- COBOL (Section 9.4)
- Fortran (Section 9.5)
- Pascal (Section 9.6)

Compiler differences fall into two categories: differences between earlier and current versions of compilers running on OpenVMS VAX, and differences between the HP versions running on the VAX, Alpha, and I64 computers. The OpenVMS I64 compilers are intended to be compatible with their OpenVMS VAX and OpenVMS Alpha counterparts. They include several qualifiers that contribute to compatibility, as described in the following sections.

The languages conform to language standards and include support for most OpenVMS VAX language extensions. The compilers produce output files with the same default file types as they do on OpenVMS VAX systems, such as .OBJ for an object module. However, some features supported by the compilers on OpenVMS VAX systems might not be available on OpenVMS I64 systems.

For more information about the compiler differences for each language, refer to the language documentation, especially the user's guides and the release notes.

9.1 Compatibility of Ada between I64 Systems and VAX Systems

Two Ada compilers are available for OpenVMS systems:

- HP Ada is provided by HP and is available on VAX and Alpha OpenVMS. It is based on the Ada 83 langauge standard and it was previously know as VAX Ada, DEC Ada, and Compaq Ada. HP Ada is not available on OpenVMS I64.
- GNAT Pro is provided by Ada Core Technologies. It is available on Alpha and I64 OpenVMS, and is based on the Ada 95 language standard. GNAT Pro is also available on many other platforms, but not VAX.

Table 9–1 compares the Ada compilers.

Table 9–1 Ada Language Support for OpenVMS

Company	Product Name	Language Standard
HP	HP Ada	Ada 83
Ada Core	GNAT Pro	Ada 95

GNAT Pro is an Ada 95 compiler, and HP Ada is an Ada 83 compiler. Generally, Ada 95 is highly upward compatible with Ada 83, so Ada 83 programs should run in Ada 95 with no changes or only minor changes.

If you use Ada on VAX and are porting your applications to I64, you need to change compilers from HP Ada to GNAT Pro.

The following options are available for port Ada applications from VAX to I64:

- Port directly to I64 using GNAT Pro
- Port to Alpha using HP Ada, then port to I64
- Port to Alpha using GNAT Pro, then port to I64

Programs that are ported must be recompiled. You cannot translate just the Ada image from VAX to I64.

For more information, see the *Ada Technical Overview and Comparison*. This document provides a detailed comparison of the compatibility of HP Ada and GNAT Pro. It is available at:

`http://h71000.www7.hp.com/commercial/ada/ada_index.html`

This document is also available on systems that have Ada installed and is located in the ADA$EXAMPLE:DEC_ADA_OVERVIEW_AND_COMPARISON.* directory.

The *GNAT Pro User's Guide* is available after installation of GNAT Pro for I64, and is located in the documentation area. It discusses compatibility with HP Ada, and on compatibility between Ada 83 and Ada 95.

For more information about GNAT Pro, see Ada Core at:

`http://www.gnat.com`

You can also contact `sales@adacore.com`.

9.1.1 Tasking Differences

Tasks on VAX are implemented differently than on Alpha and OpenVMS I64. Both Alpha and I64 use DECthreads, which is not availalbe on VAX.

9.1.2 Translating Images Using Ada

An Ada image cannot be translated to another machine target. Neither HP Ada nor GNAT Pro images can be translated to Alpha or to I64.

9.2 Compatibility of VAX BASIC and HP BASIC

HP BASIC is based on and highly compatible with VAX BASIC, which runs on VAX systems. Differences do exist, however, and these are summarized in the following sections. This information can help you develop BASIC applications that are compatible with both BASIC products and can help you migrate your VAX BASIC applications to HP BASIC on the OpenVMS I64 and Alpha operating systems.

9.2.1 VAX BASIC Features Not Available for HP BASIC

The following features of VAX BASIC are not available for I64 and Alpha systems:

- The VAX BASIC environment, which provides features specific to BASIC for program development, the RUN command, and immediate mode.
- The /SYNTAX_CHECK qualifier, which specifies syntax checking after every entered line.
- The /ANSI_STANDARD qualifier, which enforces the ANSI Minimal BASIC standard.
- The /FLAG=BP2COMPATIBILITY qualifier value, which flags uses of features in VAX BASIC programs that are not compatible with PDP-11 BASIC/PLUS2.
- The /FLAG=AXPCOMPATIBILITY qualifier value, which flags uses of features in VAX BASIC programs that are not supported by I64 BASIC/Alpha BASIC.
- The /DESIGN qualifier, which provides support for the Program Design Facility (PDF). The compiler does not attempt to compile a program when /DESIGN is specified.
- The Graphics statements, transformation functions, and other features described in the *Programming with VAX BASIC Graphics*.
- The HFLOAT VAX floating-point format for floating-point data. Also, the HFLOAT keyword is not accepted, neither in the DECLARE statement nor in various built-in functions, such as the REAL function.
- The /REAL_SIZE=HFLOAT qualifier value, which causes all floating-point data variables whose size is not explicitly declared to be HFLOAT.

9.2.2 HP BASIC Features Not Available in VAX BASIC

The following features of HP BASIC are not available in VAX BASIC:

- Support of IEEE floating-point data types SFLOAT (32-bit), TFLOAT (64-bit), and XFLOAT (128-bit), as well as the QUAD (64-bit) integer data type.
- The /INTEGER_SIZE=QUAD qualifier value, which allows the default integer data type size to be set to quadwords (that is, 64 bits).
- The /REAL_SIZE={ SFLOAT | TFLOAT | XFLOAT } qualifier values, which allow the default floating-point data type size to be set to one of the IEEE floating-point data types.
- The /SEPARATE_COMPILATION qualifier, which controls whether an individual compilation unit becomes a separate module in an object file.
- The /SYNCHRONOUS_EXCEPTIONS qualifier, which controls whether or not the compiler emits additional code to emulate VAX BASIC exception behavior.

- The /WARNINGS=ALIGNMENT qualifier, which instructs the compiler to flag all occurrences of nonnaturally aligned RECORD fields, variables within COMMONs and MAPs, and RECORD arrays.
- The /ARCHITECTURE qualifier, which allows the compiler to potentially generate more efficient code based on the target machine architecture.
- The /OPTIMIZE=(LEVEL=, TUNE=) qualifier values, which control the amount of optimization performed by the compiler.

9.2.3 VAX BASIC and HP BASIC Behavior Differences

This section describes the behavior differences between VAX BASIC and HP BASIC.

9.2.3.1 Operations with Floating-point Data Types

HP BASIC provides support for the three IEEE floating-point data types SFLOAT, TFLOAT, and XFLOAT, which are also supported by the base hardware. It is important to note that the Alpha architecture does not provide hardware support for the VAX DOUBLE or D-float data type, and the Intel Itanium architecture does not provide hardware support for any of the VAX floating-point data types (SINGLE or F-float, DOUBLE or D-float, GFLOAT, or HFLOAT).

Table 9–2 Correspondence of Floating-Point Data Types

VAX BASIC	HP BASIC (Alpha)	HP BASIC (I64)	IEEE
SINGLE (F-float)	F-float	SFLOAT	SFLOAT
DOUBLE (D-float)	GFLOAT	TFLOAT	TFLOAT
GFLOAT	GFLOAT	TFLOAT	TFLOAT
HFLOAT	XFLOAT	XFLOAT	XFLOAT

While floating-point data is stored in memory in its declared data type, the data is converted to an architecture supported type before performing operations on it. The results are then converted to the required data type before storing in memory. This conversion process might result in a loss of precision.

9.2.3.1.1 Use of (DOUBLE) D-float Data Type in HP BASIC Because the Alpha hardware does not fully support the D-float data type, HP BASIC performs BASIC DOUBLE operations (+, -, and so on) in GFLOAT; as a result, approximately 3 bits of precision is lost.

9.2.3.1.2 Use of VAX Floating-Point Data Types in HP BASIC Because the Intel Itanium hardware does not support the VAX floating-point data types, all VAX floating-point data is converted to the appropriate IEEE floating-point data type before performing any operations on it. Depending on the data type involved, some precision might be lost.

9.2.3.1.3 Implicit Use of the HFLOAT Data Type VAX BASIC performs some intermediate calculations in the HFLOAT data type, even if the source code does not explicitly specify its use. This generally occurs when mixed data-type operations are performed between large DECIMAL items and floating-point items.

HP BASIC for Alpha performs these operations in GFLOAT, and HP BASIC for I64 performs them in TFLOAT. As a result, some loss of precision is possible. The following compile-time warning message is reported if source code is encountered that results in this difference:

```
OPEPERGFL, operation performed in GFLOAT, loss of precision possible
```

9.2.3.1.4 HFLOAT Data Items in CDD Records HP BASIC maps HFLOAT data items in CDD records to a group containing a 16-byte string item, similar to other unsupported data types found in CDD records.

9.2.3.2 Default Floating-Point Data-Type Size

For VAX BASIC and HP BASIC for Alpha, the default floating-point size is SINGLE (VAX F-float) and for HP BASIC for I64, the default is SFLOAT.

9.2.3.3 Passing Parameters by Value

HP BASIC and VAX BASIC are able to pass actual parameters by value, but only HP BASIC allows by-value formal parameters.

9.2.3.4 Array Parameters

The following are differences in the way HP BASIC and VAX BASIC handle array parameters.

- Both HP BASIC and VAX BASIC perform parameter checking when an entire array is passed to a subprogram or function. When the array that was passed does not match the array that is expected by the subprogram or function, the compiler issues the error message, "Arguments don't match." VAX BASIC performs this check each time the array is referenced. HP BASIC performs this check once at the start of the subprogram or function.

  HP BASIC processes array parameters more efficiently. The following differences exist between HP BASIC and VAX BASIC in the way each processes array parameters:

  - In HP BASIC, if a subprogram or function declares an array in its parameter list, the calling program must pass an array when calling the subprogram or function. If this is not done, an unexpected failure can occur. For example, passing a null parameter instead of an array causes a memory management violation and the program fails. In VAX BASIC, it is valid for the program to pass a null parameter if the array is not accessed in the subprogram or function.
  - In HP BASIC, the subprogram cannot trap the "Arguments don't match" error. The error is signaled, but can only be trapped by the calling program.

- When passing an entire array by descriptor, VAX BASIC creates a DSC$K_CLASS_A descriptor; HP BASIC creates a DSC$K_CLASS_NCA descriptor.

  For most BASIC applications, this is not noticeable because the calling program and the called subprogram use NCA descriptors. However, a program that relies on the individual descriptor fields might need modification to work with descriptors produced by HP BASIC.

  For more information about DSC$K_CLASS_A and DSC$K_CLASS_NCA descriptors, see the *HP OpenVMS Calling Standard*.

- VAX BASIC performs no scale or precision checking when passing entire decimal arrays to a subprogram or function.

  HP BASIC subprograms and functions check all decimal arrays received by descriptor to verify that precision, scale factor, and bound information match those of the parameter in the calling program.

For example, the following program causes the error "Arguments don't match" when the subprogram test_func starts to execute:

```
DECLARE DECIMAL(5,2) a(10)
CALL test_func (a())
PRINT a(1)
END

SUB test_func (DECIMAL(10,4) b())
b(1) = 12.12
END SUB
```

- VAX BASIC performs minimal checking when receiving an array of records from a caller. For example, in the following program, VAX BASIC does not check whether the size of the array passed is equal to the size declared in the subprogram.
- HP BASIC checks that the size of the array elements are the same and that the number of dimensions match. The following program produces the error "Arguments don't match" when the subprogram test_func starts to execute:

```
RECORD rec1
   LONG a
   LONG b
END RECORD
DECLARE rec1 a(10)
CALL test_func (a())
END

SUB test_func (rec2 a())
RECORD rec2
   LONG x
   LONG y
   LONG z
END RECORD
a(2)::x = 1
END SUB
```

- VAX BASIC always performs bounds checking on arrays received as descriptor parameters.

  HP BASIC does not perform checking on arrays received as descriptor parameters if the /CHECK=NOBOUNDS qualifier is specified. In this way, arrays received as parameters are consistent with all other arrays.

9.2.3.5 DEF* Routines

In HP BASIC, DEF* routines cannot be called from within DEF routines or WHEN handlers. If such calls are attempted, the following error message is displayed:

```
BASIC-E-DEFSNOTALL, DEF* reference not allowed in DEF or handler
```

HP BASIC gives highest precedence to DEF* routines that are called from within an expression. Thus , a DEF* routine call is evaluated first. When the DEF* routine directly modifies the values of variables used within the same expression, this can affect the result of the expression. If the compiler changes the order of a DEF* call in an expression, the following warning is displayed:

```
BASIC-W-DEFEXPCOM, expression with DEF* too complex, moving <name>
invocation
```

You can avoid this error by simplifying the expression.

9.2.3.6 The /LINES Qualifier

In HP BASIC, the /LINES qualifier affects only the ERL function and determines whether BASIC line numbers are reported in run-time error messages. The following differences exist between HP BASIC and VAX BASIC:

- The /NOLINES qualifier is the default.
- You do not have to use /LINES to use the RESUME statement without a target.
- Using /LINES in programs that have line numbers on most lines can negatively affect run-time performance.

9.2.3.7 Appending Files at the DCL Command Line

VAX BASIC requires that source files used with the append operator (+) at the DCL command line contain line numbers within the files or an error message is printed.

HP BASIC does not require line numbers in either of the source files. The append operator is treated as an OpenVMS append operator. The source files are appended and compiled as if they were a single source file.

9.2.3.8 Unreachable Code Errors

HP BASIC performs extensive analysis when searching for unreachable code and might report more occurrences than VAX BASIC.

In HP BASIC, the compile-time error message for unreachable code, UNREACH, is an informational message. In VAX BASIC, the compile-time error message for unreachable code, INACOFOL, is a warning.

HP BASIC checks for DEF functions that are never referenced and displays the informational message "UNCALLED, routine xxxx can never be called."

9.2.3.9 Line Numbers

In HP BASIC, unlike VAX BASIC, does not allow duplicate line numbers or line numbers out of ascending numerical order. This restriction applies to single source files or to source files concatenated with "+" at the DCL command line. Duplicate line numbers or line numbers out of ascending order cause E level compilation errors.

HP BASIC provides an example TPU command procedure to help work around this difference. The procedure can be used to append source files and sort BASIC line numbers into ascending numerical order from one or more source files.

After installation of HP BASIC, the TPU command procedure is located in SYS$COMMON:[SYSHLP.EXAMPLES.BASIC]BASIC$ENV.TPU. Instructions for its use are in the file. Although there are no known problems, the TPU command procedure has not been thoroughly tested. As a result, it is not supported by HP.

9.2.3.10 Error-Handling Semantics

To achieve the most efficient performance, the HP BASIC compiler might reorder the execution of arithmetic instructions. Rarely does this result in error-handling semantics that are incompatible with VAX BASIC; most programs are not affected by this change.

Use the HP BASIC qualifier /SYNCHRONOUS_EXCEPTIONS for those programs that require exact VAX BASIC behavior.

9.2.3.11 Generation of Object Modules

In HP BASIC, the default behavior places all routines (SUBs, FUNCTIONs, main programs) compiled within a single source program into a single module in the object file. VAX BASIC generates each routine as a separate module. Use the HP BASIC /SEPARATE_COMPILATION qualifier to duplicate the VAX BASIC behavior.

9.2.3.12 RESUME and DEF

VAX BASIC does not enforce the documented restriction that a RESUME statement lexically outside a DEF (without a target specified) cannot resume program execution within a DEF statement. HP BASIC enforces this restriction at run time.

9.2.3.13 Exceptions

When the HP BASIC compiler determines that the result of an expression is never used, the compiler does not generate code to evaluate that expression. This causes an incompatibility with VAX BASIC if the removed expression causes an exception. In the following example, the program generates a divide-by-zero error in VAX BASIC. It runs without error in HP BASIC because HP BASIC, recognizing that the variable A is never used, does not generate the code to evaluate the expression that is assigned to A:

```
B = 5
A = B / 0
END
```

9.2.3.14 Compiler Message Differences

There is a small difference in the way HP BASIC and VAX BASIC report compiler messages. In VAX BASIC, the source information appears before the message text, and includes both source and listing line numbers. In HP BASIC, the source information appears after the message text and includes only source line numbers.

When the HP BASIC compiler reports source-line information, the message looks like this:

```
%BASIC-E-xxxxxxxxx, xxxxxxxxxxxxxx at line number YY in file xxxxxxxxxxxxx
```

In both HP BASIC and VAX BASIC, the reported line number is the physical source-line in the file. It is not the BASIC line number that might occur in the source program.

9.2.3.15 Error Status Returned to DCL

When errors occur, the HP BASIC and VAX BASIC compilers at times return a different status to DCL. For example, when the file specified at the DCL command line cannot be found, HP BASIC returns BASIC-F-ABORT; VAX BASIC returns BASIC-F-OPENIN.

9.2.3.16 SYS$INPUT

In HP BASIC, when you specify SYS$INPUT as the input file specification at the DCL command line, the object file and the listing file are named differently than in VAX BASIC. In HP BASIC, the compiler names the files with the file types .OBJ and .LIS (with no file name preceding the delimiter). In VAX BASIC, the compiler names the files NONAME.OBJ and NONAME.LIS.

9.2.3.17 FSS$ Function

The VAX BASIC compiler compiles a program that uses the FSS$ function, but if the FSS$ function is invoked at run-time, the following run-time error is displayed:

```
%BAS-F-NOTIMP, Not implemented
```

The HP BASIC compiler reports all uses of the FSS$ function by displaying the following error at compile time:

```
%BAS-E-BLTFUNNOT, built-in function not supported
```

9.2.3.18 BAS$K_FAC_NO Constant

The BAS$K_FAC_NO constant is not defined on I64 and Alpha systems. You must replace all occurrences of the EXTERNAL LONG CONSTANT BAS$K_ FAC_NO with EXTERNAL LONG CONSTANT BAS$_FACILITY. OpenVMS VAX systems use the constant BAS$K_FAC_NO to communicate the facility number between SYS$LIBRARY:BASRTL.EXE and SYS$LIBRARY:BASRTL2.EXE. The numbe is not needed on I64 and Alpha systems.

9.2.3.19 Math Functions with Different Results

Some math function results differ between HP BASIC and VAX BASIC because underlying I64 and Alpha system routines use improved algorithms to perform these operations.

9.2.3.20 Floating-Point Errors

Some programs that run successfully on VAX systems might fail on Alpha and I64 systems with division by zero or other floating-point errors. Examine your failing program for a dirty floating-point zero. A "dirty floating-point zero" is a number represented by a zero exponent and a nonzero mantissa. Most OpenVMS VAX system instructions treat the invalid floating-point number as a zero, but it causes an exception to be generated by some I64 and Alpha instructions.

You cannot create a dirty zero by using BASIC arithmetic expressions, but you can create a dirty zero by reading it from a file. BASIC I/O statements, such as GET and MOVE FROM, move bytes of data to a variable without checking whether the data is valid for the variable.

Correct the problem in one of the following ways:

- Determine how the dirty zero was created, and make the correction. This is the preferred way.
- Write a routine to clean any floating-point numbers that receive a dirty-zero value.

The following is an example of a routine that cleans a single-precision floating-point number (you can write similar routines to clean double or G-floating numbers):

```
SUB clean_single (SINGLE a)
MAP (over) SINGLE b
MAP (over) WORD w1, w2
b = a
IF (w1 AND 32640%) = 0% THEN
   a = 0
END IF
END SUB
```

The routine accepts a floating-point number, checks for a zero exponent, and clears the mantissa. It redefines the floating-point number as an integer so that the proper bits are tested.

For more information about floating-point formats and dirty zeros, see the *Alpha Architecture Reference Manual*.

9.2.3.21 Error Detection on Illegal MAT Operations

Following are two differences between HP BASIC and VAX BASIC in error detection on illegal MAT operations:

- HP BASIC correctly reports ILLOPE (Error 141 - "Illegal operation") if an attempt is made to perform matrix multiplication when the destination matrix is identical to either source matrix. VAX BASIC does not correctly detect and report the ILLOPE message if an attempt is made to perform the following matrix multiplication, where B is a virtual array and A is either a virtual array or an in-memory array:

  ```
  MAT B = A * B
  ```

- Under certain conditions, VAX BASIC does not enforce the documented restriction that arrays used in MAT operations must have zero lower bounds. HP BASIC always reports either a LOWNOTZER error at compile-time or a MATDIMERR error at run-time, when attempting to perform MAT operations on arrays with nonzero lower bounds.

9.2.3.22 Debugging Differences

There are debugging differences between VAX BASIC and HP BASIC, especially during use of the debugger STEP command around exception handlers, DEF functions, external subprograms, and GOSUB routines. These differences are described here and in the *HP BASIC for OpenVMS Systems User Manual*.

When the debugger STEP command is used in source code containing an error, differences occur in debugger behavior between OpenVMS VAX and OpenVMS BASIC/Alpha. These differences are due to architectural differences in the hardware and software of the two systems.

In HP BASIC, a STEP command at a statement that causes an exception might never return control to the debugger. The debugger cannot determine what statement in the BASIC source code will execute after the exception occurs. Therefore, set explicit breaks if you use the STEP command on statements that cause exceptions.

The following hints should help when you use the STEP command to debug programs that handle errors:

- If you use STEP at a statement that takes an error, the debugger does not regain control unless the program reaches an explicit breakpoint or the next statement that would have executed if no error had occurred. Set explicit breaks if you want the program to stop in any other place.
- Using the STEP command at a statement that takes an error does not return control to the debugger when the program reaches the error handler code. If you want the program to break when program execution enters an error handler, explicitly set a breakpoint at the error handle. This applies to both ON ERROR handlers and WHEN handlers.

- If you are within a WHEN handler, a STEP command at a statement that terminates execution within the WHEN handler (CONTINUE, RETRY, END WHEN, END HANDLER, EXIT HANDLER) does not stop unless program flow reaches a point where an explicit breakpoint is set.
- A STEP command at a RESUME statement in an ON ERROR handler stops program execution at the first line of non-error-handler code.
- Use SET BREAK/EXCEPTION at the beginning of the debugging session to prevent unexpected errors from occurring. This breakpoint is not necessary if you have set explicit breakpoints at all error handlers. However, use of this command breaks at all exceptions, thus allowing you to check whether you have the proper breakpoints to stop program execution following the exception.

9.2.3.23 Listing File Differences

Following are differences in listing files between HP BASIC and VAX BASIC:

- /MACHINE/LIST—In VAX BASIC, if you specify BASIC/MACHINE you get a listing file containing a machine language listing but no source code listing. In HP BASIC, if you specify BASIC/MACHINE, you do not get either listing. You must specify /LIST to get listing files. In HP BASIC, specifying /MACHINE/LIST gives you both the machine language and the source code in the listing file.

  When VAX BASIC creates a listing file for a program with more than one routine, it places the machine code for each routine after the source code for that routine. The listing file produced by the HP BASIC compiler contains the source listing for all the routines followed by the machine code listing for all the routines, unless you use the /SEPARATE_COMPILATION qualifier.
- %PAGE—In HP BASIC, the %PAGE directive appears on the page following the page break. In VAX BASIC, the %PAGE directive appears on the page before the page break.
- %TITLE and %SBTTL strings—These are truncated at 31 characters in HP BASIC, and 45 characters in VAX BASIC.
- Form feeds—VAX BASIC treats form feeds as %PAGE directives. HP BASIC does no special processing with form feeds. When a form feed occurs in the source file, that form feed occurs in the listing file, but no listing header information accompanies the form feed.
- /SHOW=MAP qualifier—The following differences occur in I64 BASIC/ Alpha BASIC when you use the /SHOW=MAP qualifier:
  - HP BASIC leaves the offset field in the allocation map blank in cases where the values are not applicable, or not available to the listing phase.
  - In dynamic maps of arrays, VAX BASIC reports the size of the array descriptors; HP BASIC reports the size of the array.
- Message placement—The placement of some error messages in the listing file might differ between VAX BASIC and HP BASIC. For example, in HP BASIC, errors that require flow analysis such as “unreachable code” and “routine can never be called” appear in the listing after the source code and allocation map listing. In listings for source files that contain more than one routine, these errors appear after the source and allocation listing for all routines in the compilation, unless the /SEPARATE_COMPILATION qualifier is specified.

9.2.4 Common Language Environment Differences

This section describes differences among HP BASIC, VAX BASIC, and other languages within the common language environment.

9.2.4.1 Creating PSECTs with COMMON and MAP Statements

In HP BASIC, the PSECT attributes are different from those in VAX BASIC, as shown in the following table:

HP BASIC	VAX BASIC
NOPIC	PIC
NOSHR	SHR
OCTAWORD alignment	LONG alignment

In HP BASIC, the lengths of PSECTs that the COMMON and MAP statements create are rounded up to a multiple of 16. The size of the COMMON or MAP does not change; rather the size of the PSECT does. This change is visible only to applications that use shareable images in a multilanguage environment.

Both HP BASIC and VAX BASIC create PSECTs that are compatible with those of other languages on the same platform, with the exception of MACRO. You can link with modules written in languages other than MACRO without changing code. If you link against MACRO modules that reference these PSECTs, you might need to make corresponding changes in the MACRO code.

9.2.4.2 64-Bit Floating-Point Data

In most other HP languages, the default 64-bit floating-point data type has changed from D-floating on OpenVMS VAX systems to G-floating on OpenVMS Alpha systems to T-floating on OpenVMS BASIC systems. If you communicate BASIC DOUBLE (OpenVMS D-floating) data between BASIC and one of the other languages that have made this change, you need to do one of the following:

- In the compiler command line of the other language, change the 64-bit floating-point data type to D-floating to match the behavior of Alpha BASIC, or to T-floating to match the behavior of I64 BASIC.
- In your BASIC program, change the data type of 64-bit floating-point data from DOUBLE to either GFLOAT or TFLOAT to match the other language.

9.3 Compatibility of HP C with VAX C

VAX C was the original C compiler on VAX/VMS systems. VAX C was replaced by DEC C, later rebranded as Compaq C, and more recently as HP C. If your program is written in VAX C, HP recommends that you first port to HP C on OpenVMS VAX. For details, see *Compaq C Migration Guide for OpenVMS VAX Systems*. This manual is available from the following URL:

`http://h71000.www7.hp.com/commercial/c/docs/`

If your application is already in HP C, or has been ported to HP C from VAX C, you will still face some additional issues when porting the application to I64. However, these will be the same issues that are common to all programming languages (floating point, page size, granularity, alignment, atomicity, and so on). These types of issues are covered elsewhere in this manual.

9.4 VAX COBOL and HP COBOL Compatibility and Migration

HP COBOL is based on and is highly compatible with VAX COBOL, which runs on the OpenVMS VAX system. However, there are significant differences.

For more information, see the *HP COBOL User Manual*.

9.5 Compatibility of HP Fortran on OpenVMS VAX and OpenVMS I64 Systems

This section discusses the compatibility between HP Fortran for OpenVMS I64 systems and HP Fortran 77 for OpenVMS VAX Systems (HP Fortran 77, formerly VAX FORTRAN) in the following areas:

- Language features (Section 9.5.1)
- Command-line qualifiers (Section 9.5.2)
- Interoperability with translated shared images (Section 9.5.3)
- Porting HP Fortran 77 data (Section 9.5.4)

9.5.1 Language Features

HP Fortran includes ANSI FORTRAN-77 and ISO/ANSI Fortran 9x standard features, as well as the HP Fortran 77 extensions to these Fortran standards, including:

- RECORD statement and STRUCTURE statement
- CDEC$ directives and the OPTIONS statement
- BYTE, INTEGER*1, INTEGER*2, INTEGER*4, LOGICAL*1, LOGICAL*2, LOGICAL*4
- REAL*4, REAL*8, REAL*16, COMPLEX*8, COMPLEX*16
- IMPLICIT NONE statement
- INCLUDE statement
- NAMELIST I/O
- Names up to 31 characters, including dollar sign ($) and underscore (_)
- DO WHILE and END DO statements
- Use of the exclamation point (!) for end-of-line comments
- Built-in functions %DESCR, %LOC, %REF, and %VAL
- VOLATILE statement
- DICTIONARY statement (FORTRAN-77 compiler only)
- POINTER statement data type
- Recursion
- Unformatted data conversion between disk and memory
- Indexed files
- I/O statements such as PRINT, ACCEPT, TYPE, DELETE, UNLOCK
- OPEN and INQUIRE statement specifiers, including CARRIAGECONTROL, CONVERT, ORGANIZATION, RECORDTYPE

- Other language elements identified in the appropriate Fortran language reference manuals

For detailed information about extensions and language features, see the Fortran langauge reference manual, which visually shows extensions of the FORTRAN-77 standard.

The remainder of this section summarizes language features specific to HP Fortran 77 and HP Fortran language features that are shared but interpreted differently in each language, HP Fortran restrictions that do not apply to HP Fortran 77, and data porting considerations.

9.5.1.1 Language Features Specific to HP Fortran

The following language features are available in HP Fortran but are not supported in HP Fortran 77 Version 6.4:

- Double quotation marks (" ") as delimiters for character constants. This can be disabled by specifying the /VMS qualifier.
- Naturally aligned or packed boundaries for fields of records and items in COMMON blocks.
- The INTEGER*1, INTEGER*8, and LOGICAL*8 data types.
- Support for S_floating, T_floating, and X_floating IEEE data types as well as support for nonnative unformatted data file formats, including big-endian numeric format. For a description of the native floating-point data types for Alpha systems, see the *HP OpenVMS Calling Standard*.
- LIB$ESTABLISH and LIB$REVERT are provided as intrinsic functions for compatibility with HP Fortran 77 condition handling.

  HP Fortran converts declarations to LIB$ESTABLISH to HP Fortran RTL specific entry points.
- The alternate "Z" spelling for double-precision complex intrinsic functions. (For example, the square root double-precision intrinsic function can be spelled as CDSQRT or ZSQRT.)
- The following intrinsic functions:

  IMAG
  AND
  OR
  XOR
  LSHIFT
  RSHIFT
- Certain run-time errors are specific to HP Fortran.
- Case-sensitive names.
- I/O unit numbers can be any nonnegative integer in HP Fortran. In HP Fortran 77, the values for I/O unit numbers can range from 0 to 99.

Note

When you use the HP Fortran 90 compiler, certain features associated with the ANSI/ISO Fortran 90 standard are not available in HP Fortran 77.

For an explanation of HP Fortran language features, see the Fortran language reference manual.

9.5.1.2 Language Features Specific to HP Fortran 77

The following language features are available in HP Fortran 77 but are not supported in HP Fortran:

- Automatic decomposition features of FORTRAN/PARALLEL=(AUTOMATIC)
- Manual (directed) decomposition features of FORTRAN/PARALLEL=(MANUAL) using the CPAR$ directives, such as CPAR$ DO_PARALLEL
- The following I/O and error subroutines for PDP-11 compatibility:

ASSIGN	ERRTST	RAD50
CLOSE	FDBSET	R50ASC
ERRSET	IRAD50	USEREX

  When porting existing programs, calls to ASSIGN, CLOSE, and FBDSET should be replaced with the appropriate OPEN statement. (You might consider converting DEFINE FILE statements at the same time, even though HP Fortran for OpenVMS Alpha does support the DEFINE FILE statement.)

  In place of ERRSET and ERRTST, OpenVMS condition handling might be used. Note that HP Fortran for OpenVMS Alpha supports the ERRSNS subroutine.

- Radix-50 constants in the form *n*R*xxx*

  For existing programs being ported, radix-50 constants and the IRAD50, RAD50, and R50ASC routines should be replaced by data encoded in ASCII using CHARACTER declared data.

The following HP Fortran 77 features have restricted use or are not available in HP Fortran:

- Numeric local variables are sometimes, but not always, initialized to a zero value, depending on the level of optimization used. To guarantee that a value is initialized to zero under all circumstances, use an explicit assignment or DATA statement.
- Character constants must be associated with character dummy arguments, not numeric dummy arguments. (HP Fortran 77 for OpenVMS VAX Systems passed 'A' by reference if the dummy argument was numeric.) Consider using the /BY_REF_CALL qualifier for such arguments.
- Saved dummy arrays do not work:

```
SUBROUTINE F_INIT (A, N)
REAL A(N)
RETURN
ENTRY F_DO_IT (X, I)
A (I) = X  ! No: A no longer visible
RETURN
END
```

- Hollerith actual arguments must be associated with numeric dummy (formal) arguments, not character dummy arguments.

The following language features are available in HP Fortran 77 but are not supported in HP Fortran because of differences between the Itanium architecture and the VAX architecture:

- Certain FORSYSDEF symbol definition modules might be specific to the VAX or Itanium architecture.
- Precise exception-handling control

  The handling of certain exceptions differs between VAX and I64 systems.
- REAL*16 data uses the H_floating data format on VAX systems and X_floating on I64 systems.
- VAX support for D_floating

  Because the I64 instruction set does not support the D_floating REAL*8 format, D_floating data is converted to T_floating by software during computations and then converted back to D_floating format. Thus, there are differences in D_floating arithmetic between VAX and I64 systems.

  For optimal performance on I64 systems, consider using REAL*8 data in IEEE T_floating format, perhaps using the /FLOAT qualifier to specify the format. To create an HP Fortran for OpenVMS I64 application program to convert D_floating data to T_floating format, use the file conversion methods described in the Fortran language reference manual.
- Vectorization capabilities

  Vectorization, including /VECTOR and its related qualifiers, and the CDEC$ INIT_DEP_FWD directive are not supported.

9.5.1.3 Interpretation Differences

The following language features are interpreted differently between HP Fortran 77 and HP Fortran:

- Random number generator (RAN)

  The RAN function generates a different pattern of numbers in HP Fortran than in HP Fortran 77 for the same random seed. (The RAN and RANDU functions are provided for HP Fortran 77 compatibility.)
- Hollerith constants in formatted I/O statements

  HP Fortran 77 and HP Fortran behave differently if either of the following occurs:

  – Two different I/O statements refer to the same CHARACTER PARAMETER constant as their format specifier. For example:

```
CHARACTER*(*) FMT2
PARAMETER (FMT2='(10Habcdefghij)')
READ (5, FMT2)
WRITE (6, FMT2)
```

  – Two different I/O statements use the identical character constant as their format specifier. For example:

```
READ (5, '(10Habcdefghij)')
WRITE (6, '(10Habcdefghij)')
```

  In HP Fortran 77, the value obtained by the READ statement is the output of the WRITE statement (FMT2 is ignored). However, in HP Fortran, the output of the WRITE statement is abcdefghij. (The value read by the READ statement has no effect on the value written by the WRITE statement.)

9.5.2 Command-Line Qualifiers

Although HP Fortran and HP Fortran 77 share most qualifiers, some qualifiers are specific to each platform. This section summarizes the differences between HP Fortran and HP Fortran 77 command-line qualifiers.

For complete information about the HP Fortran compilation command and options, see the *HP Fortran for OpenVMS User Manual*. For complete information about the HP Fortran 77 compilation command and options, see the *DEC Fortran User Manual for OpenVMS VAX Systems*.

To initiate compilation on either VAX or I64 systems, use the FORTRAN command. On I64 systems, use the F90 command to initiate compilation using the HP Fortran 90 compiler.

9.5.2.1 Qualifiers Specific to HP Fortran for OpenVMS I64

Table 9–3 lists HP Fortran compiler qualifiers that have no equivalent HP Fortran 77 options and are not supported in HP Fortran 77 Version 6.4.

Table 9–3 HP Fortran Qualifiers Not in HP Fortran 77

Qualifier	Description
/BY_REF_CALL	Allows character constant actual arguments to be associated with numeric dummy arguments (allowed by HP Fortran for OpenVMS VAX Systems).
/CHECK=FP_EXCEPTIONS	Controls whether messages about IEEE floating-point exceptional values are reported at run time.
/DOUBLE_SIZE	Makes DOUBLE PRECISION declarations REAL*16 instead of REAL*8.
/FAST	Sets several qualifiers that improve run-time performance.
/FLOAT	Controls the format used for floating-point data (REAL or COMPLEX) in memory, including the selection of either VAX F_floating or IEEE S_floating for KIND=4 data and VAX G_floating, VAX D_floating, or IEEE T_floating for KIND=8 data. HP Fortran 77 for OpenVMS VAX Systems provides the /[NO]G_ FLOATING qualifier.
/GRANULARITY	Controls the granularity of data access for shared data.
/IEEE_MODE	Controls how floating-point exceptions are handled for IEEE data.
/INTEGER_SIZE	Controls the size of INTEGER and LOGICAL declarations.
/NAMES	Controls whether external names are converted to uppercase, lowercase, or left as is.
/OPTIMIZE	The /OPTIMIZE qualifier supports the INLINE keyword, the LOOPS keyword, the TUNE keyword, the UNROLL keyword, and software pipelining.
/REAL_SIZE	Controls the size of REAL and COMPLEX declarations.

(continued on next page)

Table 9–3 (Cont.) HP Fortran Qualifiers Not in HP Fortran 77

Qualifier	Description
/ROUNDING_MODE	Controls how floating-point calculations are rounded for IEEE data.
/SEPARATE_COMPILATION	Controls whether the HP Fortran compiler: • Places individual compilation units as separate modules in the object file like HP Fortran 77 (/SEPARATE_COMPILATION) • Groups compilation units as a single module in the object file (/NOSEPARATE_COMPILATION, the default), which allows more interprocedure optimizations.
/SYNTAX_ONLY	Requests that only syntax checking occurs and no object file is created.
/WARNINGS	Certain keywords are not available on HP Fortran 77.
/VMS	Requests that HP Fortran use certain HP Fortran 77 conventions.

9.5.2.2 Qualifiers Specific to HP Fortran 77

This section summarizes HP Fortran 77 compiler qualifiers that have no equivalent HP Fortran qualifiers.

Table 9–4 lists compilation qualifiers specific to HP Fortran 77 Version 6.4.

Table 9–4 HP Fortran 77 Qualifiers Not Available in HP Fortran

HP Fortran 77 Qualifier	Description
/BLAS=(INLINE,MAPPED)	Specifies whether HP Fortran 77 recognizes and inlines or maps the Basic Linear Algebra Subroutines (BLAS). Available only in HP Fortran 77.
/CHECK=ASSERTIONS	Enables or disables assertion checking. Available only in HP Fortran 77.
/DESIGN=[NO]COMMENTS /DESIGN=[NO]PLACEHOLDERS	Analyzes program for design information.
/DIRECTIVES=DEPENDENCE	Specifies whether specified compiler directives are used at compilation. Available only in HP Fortran 77.
/PARALLEL=(MANUAL or AUTOMATIC)	Supports parallel processing.
/SHOW=(DATA_DEPENDENCIES,LOOPS)	Controls whether the listing file includes: • Diagnostics about loops that are ineligible for dependence analysis and data dependencies that inhibit vectorization or autodecomposition (DATA_DEPENDENCIES) • Reports about loop structures after compilation (LOOPS) The keywords DATA_DEPENDENCIES and LOOPS are available only in HP Fortran 77 for OpenVMS VAX Systems.
/VECTOR	Requests vector processing. Available only in HP Fortran 77.

(continued on next page)

Table 9–4 (Cont.) HP Fortran 77 Qualifiers Not Available in HP Fortran

HP Fortran 77 Qualifier	Description
/WARNINGS=INLINE	Controls whether the compiler prints informational diagnostic messages when it is unable to generate inline code for a reference to an intrinsic routine. Available only in HP Fortran 77.

All CPAR$ directives and certain CDEC$ directives associated with directed (manual) decomposition and their associated qualifiers or keywords are specific to HP Fortran 77, as described in the *DEC Fortran Language Reference Manual*.

For details about the HP Fortran 77 compilation commands and options, see the *DEC Fortran User Manual for OpenVMS VAX Systems*.

9.5.3 Interoperability with Translated Shared Images

Using HP Fortran, you can create images that can interoperate with translated images at image activation (run time).

To allow the use of translated shared images:

- On the FORTRAN or F90 command line, specify the /TIE qualifier.
- On the LINK command line, specify the /NONATIVE_ONLY qualifier.

The created executable image contains code that allows the resulting executable image to interoperate with shared images, including allowing the HP Fortran 77 RTL (FORRTL) to work with the HP Fortran RTL (DEC$FORTRTL). The native (HP Fortran RTL) and translated (HP Fortran 77 RTL) programs can perform I/O to the same unit number, as long as the RTL that opens the file also closes it.

Programs should use the intrinsic names (without the prefix) rather than calling routines by their complete (*fac*$*xxxx*) name. One allowable exception to using *fac$xxxx* names is that translated image programs declare the FOR$RAB system function as EXTERNAL. Native I64 programs should use FOR$RAB as an intrinsic function.

9.5.4 Porting HP Fortran 77 Data

Record types are identical for Digial Fortran 77 and HP Fortran. If needed, transport the data using the EXCHANGE command with the /NETWORK and /TRANSFER=BLOCK qualifiers. To convert the file to Stream_LF format during the copy operation, use /TRANSFER=(BLOCK,RECORD_SEPARATOR=LF) instead of /TRANSFER=BLOCK, or specify the /FDL qualifier to the EXCHANGE command to change the record type or other file characteristics.

If you need to convert unformatted floating-point data, keep in mind that HP Fortran 77 programs (VAX hardware) store REAL*4 or COMPLEX*8 data in F_floating format, REAL*8, REAL*16, or COMPLEX*16 data in either D_floating or G_floating format, and REAL*16 data in H_floating format. HP Fortran for OpenVMS Alpha programs (running on Alpha hardware) store REAL*4, REAL*8, REAL*16, COMPLEX*8, and COMPLEX*16 data in one of the formats shown in Table 9–5.

Table 9–5 Floating-Point Data on VAX and I64 Systems

Data Declaration	VAX Formats	I64 Formats
REAL*4 and COMPLEX*8	VAX F_floating format	IEEE S_floating or VAX F_floating[1] format
REAL*8 and COMPLEX*16	VAX D_floating or G_ floating format	IEEE T_floating, VAX D_floating[1], or VAX G_floating format
REAL*16	VAX H_floating	X_floating. Requires conversion, perhaps using the /CONVERT qualifier or associated OPTION statement, logical name, or OPEN statement /CONVERT keyword. You can also use the RTL routine CVT$CONVERT_FLOAT.

[1]On I64 systems, use of VAX F_floating, D_floating, or G_floating formats involving many computations is not recommended. Consider converting F_floating format to IEEE S_floating format, and converting D_floating and G_floating formats to IEEE T_floating format in a conversion program that uses the HP Fortran for OpenVMS I64 conversion routines.

9.6 Compatibility of HP Pascal for I64 Systems with HP Pascal for VAX Systems

This section compares HP Pascal to other HP Pascal compilers and lists the differences between HP Pascal on VAX and I64 systems. For a complete description of these features, see the *HP Pascal for OpenVMS Language Reference Manual*.

9.6.1 Unused External Symbols

On OpenVMS VAX, the HP Pascal compiler produces global symbol definitions (GSDs) only for symbols present in the final instruction stream. On OpenVMS I64 systems, HP Pascal produces GSDs for all external definitions regardless of whether they were used in the program or not. This might require that additional objects be present when linking or additional shareable images be present when running certain programs.

Such programs can be reorganized to declare such external definitions only when they are actually used in the program.

9.6.2 Sharing Environment Files Across Platforms

The compiler only inherits environment files created from a compiler for the same target platform. For example, you cannot inherit environment files generated by HP Pascal for OpenVMS VAX with the HP Pascal for OpenVMS I64 compiler.

9.6.3 Default Size for Enumerated Types and Booleans

The default size for enumerations and Booleans in unpacked structures is longword on I64 systems. On VAX systems, the default is a byte for Booleans and small enumerations or word for larger enumerations. If you need the VAX behavior on I64 systems, you can use the /ENUMERATION_SIZE=BYTE qualifier, the [ENUMERATION_SIZE(BYTE)] attribute, or you can place individual [BYTE] or [WORD] attributes on the affected fields or components. The default for OpenVMS VAX compilers remains /ENUMERATION_SIZE=BYTE for compatibility.

9.6.4 Default Data Layout for Unpacked Arrays and Records

On I64 systems, the default data layout is natural alignment. This means that record fields and array components are aligned on boundaries based on their size. On VAX systems, the default alignment rule is to allocate such fields on the next byte boundary. If you need the VAX behavior on I64 systems, you can use the /ALIGN=VAX qualifier or the [ALIGN(VAX)] attribute.

9.6.5 Default Floating Format

On I64 systems, the default floating-point format is IEEE format. The compiler uses IEEE S_floating for REAL, IEEE T_floating for DOUBLE, and X_floating for QUADRUPLE datatypes. On VAX systems, the default floating-point format is VAX format. If you need the VAX floating format on OpenVMS I64 systems, you can use the /FLOAT=G_FLOAT or /FLOAT=D_FLOAT qualifier or the [FLOAT()] module-level attribute. When VAX format is requested on I64 systems, the compiler converts each VAX format value to IEEE format before performing any operation. The compiler then converts the value back to VAX format before storing it back to memory. Because of differences in the VAX and IEEE formats, some slight differences in precision and accuracy might occur.

9.6.6 IADDRESS and VOLATILE

The IADDRESS built-in assumes that its parameter is a VOLATILE variable, a VOLATILE parameter, or a routine entry point. Unlike the ADDRESS built-in, the IADDRESS buil-in does not issue a warning if the parameter does not have the VOLATILE attribute.

On VAX systems, the HP Pascal compiler often allocates variables so that they exist for the entire routine in which they were declared. In these situations, using the IADDRESS built-in to obtain the address of the variable works as expected. Usually, the address is passed to a system service by way of an item list or something similar.

On I64 systems, the HP Pascal compiler is much more agressive with optimizating data layout on the stack. In the absence of a VOLATILE attribute, the compiler allocates variables for the smallest possible duration. If the address is taken with IADDRESS, by the time the address is written into by a system service, the variable might no longer exist and the memory store would corrupt another variable.

In summary, if the IADDRESS built-in is used on automatic variables or parameters, then the VOLATILE attribute must be used to ensure proper behavior.

9.6.7 INT on Large Unsigned Numbers Now Overflows

On VAX systems, the INT built-in accepts large unsigned numbers and silently converts them to negative integers. During the addition of 64-bit integer types to HP Pascal, it became apparent that this behavior was wrong. Now, when overflow checking is enabled, the INT built-in signals a run-time error if its actual parameter cannot be represented as an INTEGER32 value.

If you have a large unsigned value that you want to convert to a negative integer, you must use a typecast to perform the operation.

9.6.8 Bound Procedure Values

On VAX systems, a bound procedure value is a 2-longword data structure that holds the address of the entry point and a frame-pointer to define the nested environment. HP Pascal expects one of these 2-longword structures for PROCEDURE or FUNCTION parameters. Additionally, a called routine needs to identify when it receives a bound procedure value and not a simple routine address. When passing a routine to a %IMMED formal routine parameter, HP Pascal passes the address of the entry point.

On I64 systems, a bound procedure value is just a special type of function descriptor which invokes a hidden jacket routine which in turns initializes the frame-pointer and calls the real routine. Given this structure, a routine that is calling another routine indirectly does not need to do anything special for bound procedure values. Likewise, passing routines by %IMMED (or asking for the IADDRESS of a routine) passes the address of a function descriptor just as if the %IMMED was not present. There is no direct way in HP Pascal to obtain the actual code address of a routine because the routine is not generally useful without the associated function descriptor.

9.6.9 Different Descriptor Classes for Conformant Array Parameters

For conformant parameters, HP Pascal uses the "by descriptor" mechanism to pass them from one routine to another. For conformant array parameters, HP Pascal uses a CLASS_A descriptor on VAX systems and a CLASS_NCA descriptor on I64 systems. The CLASS_NCA descriptors generate more efficient code when accessing array components. If you have a foreign routine that constructs CLASS_A descriptors for Pascal, you need to examine the code to see if changes are necessary. For certain actual parameters, the CLASS_A and CLASS_NCA descriptors are identical except for the DSC$B_CLASS field (which HP Pascal does not examine). For other parameters, you either must generate a CLASS_ NCA descriptor or add an explicit CLASS_A attribute to the formal conformant parameter in the Pascal routine.

9.6.10 Pascal Features Not Available on OpenVMS I64

The following features from HP Pascal for OpenVMS VAX systems that are not available on HP Pascal for OpenVMS I64:

- The MFPR and MTPR built-in routines
- The /DESIGN=COMMENTS DCL qualifier
- The /SHOW=TABLE_OF_CONTENTS DCL qualifier and listing section
- The /SHOW=INLINE DCL qualifier and listing section

There are no alternatives or workarounds for these features.

Before migrating from OpenVMS VAX to OpenVMS I64, you can compile your programs with the /PLATFORM=OPENVMS_AXP qualifier. You must use the OPENVMS_AXP keyword because the currently shipping HP Pascal compiler for OpenVMS VAX systems does not recogize the OPENVMS_I64 keyword for the /PLATFORM DCL qualifier. The issues for porting from OpenVMS VAX to OpenVMS I64 are essentially the same as those from OpenVMS VAX to OpenVMS Alpha, with the only difference being the default floating format.

9.6.11 Pascal Record Layout Guide

The HP Pascal kit provides a document that further describes data layout and data conversion issues for programs migrating from OpenVMS VAX to OpenVMS I64. The file is located at SYS$HELP:PASCAL_RECORD_LAYOUT_GUIDE.MEM.

A

Application Evaluation Checklist

Development History and Plans

1. Does the application currently run on other operating systems or hardware architectures? ☐ YES ☐ NO

   If yes, does the application currently run on a RISC system? ☐ YES ☐ NO

   [If so, migrating to OpenVMS I64 it will be easier.]

2. What are your plans for the application after migration?

   a. No further development ☐ YES ☐ NO

   b. Maintenance releases only ☐ YES ☐ NO

   c. Additional or changed functionality ☐ YES ☐ NO

   d. Maintain separate VAX and I64 sources ☐ YES ☐ NO

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

External Dependencies

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

   [This helps you plan for the resources needed for migration.]

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

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

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

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

6. a. What version of OpenVMS does your application currently run on? _____

   b. Does the application run on OpenVMS VAX Version 6.1? ☐ YES ☐ NO

   c. Does the application use features that are not available on OpenVMS I64? ☐ YES ☐ NO

   [The migration base for OpenVMS I64 is OpenVMS VAX Version 6.1. If you answer YES to question c, your application might use features that are not yet supported on OpenVMS I64, or it might be linked against an OpenVMS RTL or other shareable image that is incompatible with the current version of OpenVMS I64.]

7. Does the application require layered products to run?

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

   b. From third parties: ☐ YES ☐ NO

   [If you answer YES to item a and are uncertain whether the HP layered products are yet available for OpenVMS I64, check with a HP support representative. If you answer YES to item b, check with your third-party supplier.]

Composition of the Application

8. How large is your application?

   a. How many modules? _____

   b. How many lines or kilobytes of code? _____

   c. How much disk space is required? _____

   [This helps you "size" the effort and the resources required for migration.]

9. a. Do you have access to all source files that make up your application? ☐ YES ☐ NO

   b. If you are considering using HP Services, will it be possible to give HP access to these source files and build procedures? ☐ YES ☐ NO

   [If you answer YES to question a, translation may be your only migration option for the files with missing sources. A YES answer to question b allows you to take advantage of a greater range of HP migration services.]

10. a. What languages is the application written in? (If multiple languages are used, give the percentages of each.) _____

    [If the compilers are not yet available, you must translate or rewrite in a different language.]

    b. If you use VAX MACRO, what are your specific reasons? _____

c. Can the function of the VAX MACRO code be performed by a high-level-language compiler or a system service (such as $GETJPI for retrieving process names)? ☐ YES ☐ NO

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

b. If yes, do they require DEC Test Manager? ☐ YES ☐ NO

[If you answer YES to question a, consider migrating those regression tests. The DEC Test Manager is not available at the initial release of OpenVMS I64. Contact an HP support representative if your regression tests depend on the DEC Test Manager.]

Dependencies on the VAX Architecture

12. a. Does the application use the H_floating data types? ☐ YES ☐ NO

b. Does the application use the D_floating data types? ☐ YES ☐ NO

c. If the application uses D_floating, does it require 56 bits of precision (16 decimal digits) or would 53 bits (15 decimal digits) suffice? ☐ 56 bits ☐ 53 bits

[If you answer YES to question a, you must either translate your application to obtain H_floating compatibility, or convert the data to G_floating, S_floating, or T_floating format. If you answer YES to question b, you must either translate the application to obtain full 56-bit VAX precision D_floating compatibility, accept the 53-bit precision D_ floating format provided by I64 systems, or convert the data to G_floating, S_floating, or T_floating format.]

13. a. Does the application use large amounts of data or data structures? ☐ YES ☐ NO

b. Is the data byte, word, or longword aligned? ☐ YES ☐ NO

[If you answer YES to question a, but NO to question b, consider aligning your data naturally to achieve optimal I64 performance. You must align data naturally if the data is in a global section shared among a number of processes, or if it is shared between a main program and an AST.]

14. Does the application make assumptions about how compilers align data (that is, does the application assume that data structures are packed, aligned naturally, aligned on longwords, and so on)? ☐ YES ☐ NO

[If you answer YES, you should consider portability and interoperability issues resulting from differences in compiler behavior, both on the I64 platform and between the VAX and I64 platforms. Be aware that compiler defaults for data alignment vary, as do compiler switches for forcing alignment. Typically, VAX systems default to a packed style alignment, whereas I64 compilers default to natural alignment where possible.]

15. a. Does the application assume a 512-byte page size? ☐ YES ☐ NO

b. Does the application assume that a memory page is the same size as a disk block? ☐ YES ☐ NO

[If you answer YES to question a, be prepared to adapt the application to accommodate the I64 page size, which is much larger than 512 bytes and varies from system to system. Avoid hardcoded references to the page size; rather, use memory management system services and RTL routines wherever possible. If you answer YES to question b, you should examine all calls to the $CRMPSC and $MGBLSC system services that map disk sections to memory and remove these assumptions.]

16. Does the application call OpenVMS system services? ☐ YES ☐ NO

Does the application call services that:

a. Create or map global sections (such as $CRMPSC, $MGBLSC, $UPDSEC) ☐ YES ☐ NO

b. Modify the working set (such as $LCKPAG, $LKWSET) ☐ YES ☐ NO

c. Manipulate virtual addresses (such as $CRETVA, $DELTVA) ☐ YES ☐ NO

[If you answer YES to any of these, you might need to examine your code to make sure that it specifies the required input parameters correctly.]

17. 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 helps you determine whether you 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 migrate an application most easily.]

18. 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.]

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

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

20. a. Does the application contain condition handlers? ☐ YES ☐ NO

    b. Does the application rely on immediate reporting of arithmetic exceptions? ☐ YES ☐ NO

    [The Intel Itanium architecture does not provide immediate reporting of arithmetic exceptions. If your handler attempts to fix the condition and restart the instruction sequence that led to the exception, you need to alter the handler.]

21. Does the application run in privileged mode or link against SYS.STB? ☐ YES ☐ NO

    If so, why? _____

    [If your application links against the OpenVMS executive or runs in privileged mode, you must rewrite it to work as a native I64 image.]

22. Do you write your own device drivers? ☐ YES ☐ NO

23. Does the application use connect-to-interrupt mechanisms? ☐ YES ☐ NO

    If yes, with what functionality? _____

24. Does the application create or modify machine instructions? ☐ YES ☐ NO

    [Guaranteeing correct execution of instructions written to the instruction stream requires great care on OpenVMS I64.]

25. What parts of the application are most sensitive to performance? I/O, floating point, memory, real time (that is, interrupt latency, and so on). _____

    [This helps you determine how to prioritize work on the various parts of your application and allows HP to plan performance enhancements that are most meaningful to customers.]

Glossary

alignment

See *natural alignment*.

atomic instruction

An instruction that consists of one or more discrete operations that are handled by the hardware as a single operation, without interruption.

atomic operation

An operation that cannot be interrupted by other system events, such as an AST (asynchronous system trap) service routine; an atomic operation appears to other processes to be a single operation. Once an atomic operation starts, it always completes without interruption.

Read-modify-write operations are typically not atomic at an instruction level on a RISC machine.

byte granularity

A property of memory systems in which adjacent bytes can be written concurrently and independently by different processes or processors.

CISC

See *complex instruction set computer*.

compatibility

The ability of programs written for one type of computer system (such as OpenVMS VAX) to execute on another type of system (such as OpenVMS I64).

complex instruction set computer (CISC)

A computer that has individual instructions that perform complex operations, including complex operations performed directly on locations in memory. Examples of such operations include instructions that do multibyte data moves or substring searches. CISC computers are typically contrasted with *RISC* (*reduced instruction set computer*) computers.

concurrency

Simultaneous operations by multiple agents on a shared object.

granularity

A characteristic of storage systems that defines the amount of data that can be read or written with a single instruction, or read or written independently. VAX systems have byte or multibyte granularities while disk systems typically have 512-byte or greater granularities.

image section

A group of program sections with the same attributes (such as read-only access, read/write access, absolute, relocatable, and so on) that is the unit of virtual memory allocation for an image.

interlocked instruction

An instruction that performs some action in a way that guarantees the complete result as a single, uninterruptible operation in a multiprocessing environment. Since other potentially conflicting operations can be blocked while the interlocked instruction completes, interlocked instructions can have a negative performance impact.

load/store architecture

A machine architecture in which data items are first loaded into a processor register, then operated on, and finally stored back to memory. No operations on memory other than load and store are provided by the instruction set.

longword

Four contiguous bytes (32 bits) starting on any addressable byte boundary. Bits are numbered from right to left, 0 to 31. The address of the longword is the address of the byte containing the low-order bit (bit 0). A longword is *naturally aligned* if its address is evenly divisible by 4.

multiple instruction issue

Issuing more than one instruction during a single clock cycle.

natural alignment

Data storage in memory such that the address of the data is evenly divisible by the size of the data in bytes. For example, a naturally aligned longword has an address that is evenly divisible by 4, and a naturally aligned quadword has an address that is evenly divisible by 8. A structure is naturally aligned when all its members are naturally aligned.

page size

The number of bytes that a system's hardware treats as a unit for address mapping, sharing, protection, and movement to and from secondary storage.

pagelet

A 512-byte unit of memory in an Itanium environment. On OpenVMS I64 systems, certain DCL and utility commands, system services, and system routines accept as input or provide as output memory requirements and quotas in terms of pagelets. Although this allows the external interfaces of these components to be compatible with those of VAX systems, OpenVMS I64 internally manages memory only in even multiples of the CPU memory page size.

PALcode

See *privileged architecture library*.

privileged architecture library (PAL)

A library of callable routines for performing instructions unique to a particular operating system. Special instructions call the routines, which must run without interruption.

processor status (PS)

On Alpha systems, a privileged processor register consisting of a *quadword* of information including the current access mode, the current interrupt priority level (IPL), the stack alignment, and several reserved fields.

processor status longword (PSL)

On VAX systems, a privileged processor register consisting of a word of privileged processor status and the *processor status word* itself. The privileged processor status information includes the current interrupt priority level (IPL), the previous access mode, the current access mode, the interrupt stack bit, the trace trap pending bit, and the compatibility mode bit.

processor status word (PSW)

On VAX systems, the low-order word of the *processor status longword*. Processor status information includes the condition codes (carry, overflow, 0, negative), the arithmetic trap enable bits (integer overflow, decimal overflow, floating underflow), and the trace enable bit.

program counter (PC)

That portion of the CPU that contains the virtual address of the next instruction to be executed. Most current CPUs implement the program counter as a register. This register is visible to the programmer through the instruction set.

quadword

Four contiguous words (64 bits) starting on any addressable byte boundary. Bits are numbered from right to left, 0 to 63. The address of a quadword is the address of the word containing the low-order bit (bit 0). A quadword is *naturally aligned* if its address is evenly divisible by 8.

quadword granularity

A property of memory systems in which adjacent *quadwords* can be written concurrently and independently by different processes or processors.

read-modify-write operation

A hardware operation that involves the reading, modifying, and writing of a piece of data in main memory as a single, uninterruptible operation.

read-write ordering

The order in which memory on one CPU becomes visible to an execution agent (a different CPU or device within a tightly coupled system).

reduced instruction set computer (RISC)

A computer that has an instruction set reduced in complexity, but not necessarily in the number of instructions. RISC architectures typically require more instructions than *CISC* architectures to perform a given operation, because an individual instruction performs less work than a CISC instruction.

RISC

See *reduced instruction set computer*.

synchronization

A method of controlling access to some shared resource so that predictable, well-defined results are obtained when operating in a multiprocessing environment or in a uniprocessing environment using shared data.

translated code

The native OpenVMS I64 object code in a translated image. Translated code includes:

- OpenVMS I64 code that reproduces the behavior of equivalent VAX code in the original image
- Calls to the *Translated Image Environment (TIE)*

translated image

An OpenVMS I64 executable or shareable image created by *translation* of the object code of a VAX image. The translated image, which is functionally equivalent to the VAX image from which it was translated, includes both *translated code* and the original image. See *VAX Environment Software Translator*.

Translated Image Environment (TIE)

A native Alpha shareable image that supports the execution of *translated images*. The TIE processes all interactions with the native Alpha system and provides an environment similar to OpenVMS VAX for the translated image by managing VAX state; by emulating VAX features such as exception processing, AST delivery, and complex VAX instructions; and by interpreting untranslated VAX instructions.

translation

The process of converting a VAX binary image to an OpenVMS I64 image that runs with the assistance of the TIE on an Alpha system. Translation is a static process that converts as much VAX code as possible to native Alpha instructions. The TIE interprets any untranslated VAX code at run time.

VEST

See *VAX Environment Software Translator*.

VAX Environment Software Translator (VEST)

A software migration tool that performs the *translation* of VAX executable and shareable images into translated images that run on Alpha systems. See *translated image*.

word granularity

A property of memory systems in which adjacent words can be written concurrently and independently by different processes or processors.

writable global section

A data structure (for example, FORTRAN global common) or shareable image section potentially available to all processes in the system for use in communicating between processes.

Index

A

B

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X