Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications

Order Number: AA–PV63A–TE

May 1993

This manual describes how to create an OpenVMS AXP version of an OpenVMS VAX application.

Revision/Update Information: This is a new manual.

Software Version: OpenVMS AXP Version 1.5
OpenVMS VAX Version 6.0

Digital Equipment Corporation
Maynard, Massachusetts

May 1993

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Preface

Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications is designed to assist developers in moving OpenVMS VAX applications to an OpenVMS AXP system. The manual consists of the following chapters:

- Chapter 1 provides an overview of areas to look at to uncover VAX dependencies in your application.
- Chapter 2 describes how to handle dependencies your application may have on the VAX page size.
- Chapter 3 describes how to handle dependencies your application may have on the synchronization provided by the VAX architecture with regard to data access by multiple processes.
- Chapter 4 describes the implications of data declarations on an AXP system, including alignment concerns.
- Chapter 5 describes how to handle dependencies your application may contain on the VAX condition handling facility.
- Chapter 6 describes how to create native AXP images that can call and be called by translated VAX images.
- Appendix A contains brief summaries of the new and changed features supported by the Ada, C, COBOL, FORTRAN, and Pascal programming languages on AXP systems.

Intended Audience

This manual is intended for experienced software engineers responsible for moving application code written in high- or mid-level programming languages such as C or FORTRAN.

Associated Documents

This manual is part of a set of manuals that describe various aspects of migrating OpenVMS VAX applications to an OpenVMS AXP system. The other manuals in this set are as follows:

- *Migrating to an OpenVMS AXP System: Planning for Migration* provides an overview of the VAX to Alpha AXP migration process and information to help you plan a migration. It discusses the decisions you must make in planning a migration and the ways to get the information you need to make those decisions. In addition, it describes the migration methods available so that you can estimate the amount of work required for each method and select the method best suited to a given application.

- *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code* describes how to port VAX MACRO code to an AXP system using the MACRO–32 compiler for OpenVMS AXP. It describes the features of the compiler, presents a methodology for porting VAX MACRO code, identifies nonportable coding practices, and recommends alternatives to such practices. The manual also provides a reference section with detailed descriptions of the compiler's qualifiers, directives, and built-ins, and the system macros created for porting to AXP systems.

In addition, the *DECmigrate for OpenVMS AXP Systems Translating Images* manual describes the VAX Environment Software Translator (VEST) utility. This manual is distributed with the optional layered product, DECmigrate for OpenVMS AXP, which supports the migration of OpenVMS VAX applications to OpenVMS AXP systems. The manual describes how to use VEST to convert most user-mode VAX images to translated images that can run on AXP systems; how to improve the run-time performance of translated images; how to use VEST to trace AXP incompatibilities in an 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.

Conventions

In this manual, every use of OpenVMS AXP means the OpenVMS AXP operating system, every use of OpenVMS VAX means the OpenVMS VAX operating system, and every use of OpenVMS means both the OpenVMS AXP operating system and the OpenVMS VAX operating system.

The following conventions are 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.
$\boxed{\text{Return}}$	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.)
. . .	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 format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.

[ ]	In format descriptions, brackets indicate optional elements. You can choose one, none, or all of the choices. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification or in the syntax of a substring specification in an assignment statement.)
{ }	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason. Boldface text is also used to show user input in Bookreader versions of the book.
italic text	Italic text emphasizes important information, indicates variables, and indicates complete titles of manuals. Italic text also represents information that can vary in system messages (for example, Internal error *number*), command lines (for example, /PRODUCER=name), and command parameters in text.
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
-	A hyphen in code examples indicates that additional arguments to the request are provided on the line that follows.
numbers	All numbers in text are assumed to be decimal, unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1 Introduction

This chapter introduces the general process of moving an application that runs on a VAX system to an AXP system by recompiling and relinking the source files that make up the application. Specifically, this chapter covers the following topics:

- Using AXP versions of the tools in the VAX programming environment, such as native compilers and the linker
- Identifying dependencies your application may have on aspects of the VAX architecture

1.1 Overview

In general, if your application is written in a high-level programming language, you should be able to get it running on an AXP system with a minimum of effort. High-level languages insulate applications from dependence on the underlying machine architecture. In addition, for the most part, the programming environment on AXP systems duplicates the programming environment on VAX systems. Using native AXP 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 AXP image.

However, it is possible to introduce architectural dependencies even in applications written in high-level languages. The following sections describe the programming environment on an AXP system and provide guidelines for identifying code in your application source files that may not be able to be moved to an AXP system without modification.

1.2 Recompiling Your Application with Native AXP Compilers

Many of the languages supported on VAX systems are supported on AXP systems, such as FORTRAN and C. For complete information about the availability of programming languages on AXP systems, see *Migrating to an OpenVMS AXP System: Planning for Migration*.

The compilers available on AXP 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.

Note, however, that some features supported by the compilers on VAX systems may not be available in the same compiler on AXP systems. In addition, some compilers on AXP systems support new features not supported by their counterparts on VAX systems. To provide compatibility, some compilers support compatibility modes. For example, the DEC C for OpenVMS AXP systems compiler supports a VAX C compatibility mode that is invoked by specifying the

/STANDARD=VAXC qualifier. Appendix A lists the features of several compilers available on both the VAX and AXP systems.

1.3 Identifying Dependencies on the VAX Architecture in Your Application

Even if your application recompiles successfully with a compiler that generates native AXP code, it may still contain subtle dependencies on the VAX architecture. The operating system has been designed to provide a high degree of compatibility; however, the fundamental differences between the two architectures inevitably create certain inconsistencies. The following list highlights those areas of your application you should examine. The remaining chapters in this manual provide more information about these topics.

- Check the data declarations contained in your application. The high-level language data types you selected to represent data items on a VAX system may not be the best choice on an AXP system. In particular, consider the following:
  - **Data packing**—Applications on VAX systems typically use the smallest available data type to represent a data item to achieve efficient use of memory resources. For various reasons (described in Chapter 4), using larger data types may be more efficient on AXP systems.
  - **Data-type selection**—The Alpha AXP architecture supports most of the VAX native data types; however, certain VAX data types, such as the H_float floating-point data type, are not supported. Check to see if your application depends on the size or bit representation of an underlying native data type. Chapter 4 contains a list of all the data types supported by the Alpha AXP architecture.
  - **Shared access to data**—Check any data item that is accessed by multiple threads of execution. The VAX architecture includes instructions that can perform certain complex operations, such as incrementing a variable, that appear as a single, noninterruptable operation to other threads of execution. The Alpha AXP architecture is a load-store architecture that does not support atomic memory-to-memory modifications. Chapter 3 provides more information about this topic.

    In addition, the VAX architecture supports instructions that can manipulate byte- and word-sized data in a single noninterruptable operation. The Alpha AXP architecture supports noninterruptable access only to aligned longword- or aligned quadword-sized data. Chapter 3 and Chapter 4 describe how this can affect your application.
  - **Buffer size**—Your application may determine the size of certain data buffers based on the VAX page size. Different implementations of the Alpha AXP architecture can support 8K, 16K, 32K, or 64K byte pages. Search your application for the text strings “512” and “511” (or the hexadecimal equivalent “200”) to find dependencies on the VAX page size. Chapter 2 describes how to adapt your application to this change in page size.
- Check any condition handlers your application may include. While the condition handling facility on AXP systems is functionally equivalent to the VAX condition handling facility, certain aspects of the facility have changed, such as the format of the mechanism array. In addition, the way in which

arithmetic exceptions are reported has changed. For information about this topic, see Chapter 5.

- Check for dependence on the AST parameter list. While the AST parameter list on AXP systems has the same format as on VAX systems, only the AST parameter field can be used. The other fields in the AST parameter list (contents of R0, R1, program counter [PC], and processor status [PS]) are provided for compatibility only and have no subsequent use after the AST procedure exits.

1.4 Relinking Your Application on an AXP System

Once you successfully recompile your source files, you must relink your application to create a native AXP 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 AXP systems, you will probably need to modify the LINK command used to build your application. The following list describes some of these linker changes that may affect your application's build procedure. See the *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 AXP systems, instead of creating a transfer vector file, you must declare universal symbols in a linker options file by specifying the 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 AXP systems, you link against the OpenVMS executive by specifying the /SYSEXE qualifier.
- **Optimizing the performance of images**—On AXP systems, the linker can perform certain optimizations that can improve the performance of the images it creates. In addition, the linker can create shareable images that can be installed as resident images, another performance enhancement.
- **Processing shareable images implicitly**—On VAX systems, when you specify a shareable image in a link operation, the linker also processes all the shareable images to which that shareable image was linked. On AXP systems, to include these shareable images in your build procedure, you must explicitly specify them.

The linker supports several qualifiers and options, listed in Table 1–1, that are specific to AXP systems. The table also lists those linker qualifiers supported on VAX systems that are not supported by the linker on AXP systems.

Table 1–1 Linker Qualifiers and Options Specific to AXP Systems

Qualifiers	Description
/DEMAND_ZERO	Controls how the linker creates demand-zero image sections.
/GST	Directs the linker to create a global symbol table (GST) for a shareable image (the default). More typically specified as /NOGST when used to create shareable images for run-time kits.
/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 /NONATIVE_ONLY is specified during linking, the image activator uses the PSB information, if any, provided in the object modules specified as input files to the link operation to create jacket routines. Jacket routines are necessary to allow native AXP images to work with translated VAX images.
/REPLACE	Directs the linker to perform certain optimizations that can improve the performance of the image it is creating, when requested to do so by the compilers (the default).
/SECTION_BINDING	Directs the linker to create a shareable image that can be installed as a resident image.
/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
BASE= option	Not supported on AXP systems.
DZRO_MIN= option	Not supported on AXP systems.
ISD_MAX= option	Not supported on AXP systems.
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 AXP shareable images.
UNIVERSAL= option	Not supported on AXP systems.

1.5 Compatibility Between the Mathematics Libraries Available on VAX and AXP Systems

Mathematical applications using the standard VMS call interface to the OpenVMS Mathematics (MTH$) Run-Time Library need not change their calls to MTH$ routines when migrating to an AXP system. Jacket routines are provided that map MTH$ routines to their math$ counterparts in the Digital Portable Mathematics Library (DPML) for AXP 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 and you should expect to see small differences in the precision of the mathematical results.

To maintain compatibility with future libraries and to create portable mathematical applications, Digital recommends that you use the DPML routines available through the high-level language of your choice (for example, FORTRAN or C) rather than using the call interface. Significantly higher performance and accuracy are also available to you with DPML routines.

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

1.6 Determining the Host Architecture

Your application may need to determine whether it is running on an OpenVMS VAX system or an AXP system. From within your program, you can obtain this information by calling the $GETSYI system service (or the LIB$GETSYI RTL routine), 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 AXP system, the $GETSYI system service returns the value 2.

Example 1–1 illustrates 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 in an application, see Section 2.4, where it is used to obtain the page size of an AXP system.)

Example 1–1 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_AXP
$ ON_VAX:
$ !
$ ! Do VAX-specific processing
$ !
$ exit
$ ON_ALPHA_AXP:
$ !
$ ! Do Alpha AXP-specific processing
$ !
$ exit
```

Note, however, that the ARCH_TYPE item code is available only on VAX systems running 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 1–2.

Table 1–2 $GETSYI Item Codes That Specify Host Architecture

Keyword	Usage
ARCH_TYPE	Returns 1 on a VAX system; returns 2 on an AXP system. Supported on AXP systems and on VAX systems running OpenVMS Version 5.5 or later.
ARCH_NAME	Returns text string “VAX” on VAX machines and text string “Alpha” on AXP machines. Supported on AXP systems and on VAX systems running OpenVMS Version 5.5 or later.
HW_MODEL	Returns an integer that identifies a particular hardware model. All values equal to or larger than 1024 identify AXP systems.
CPU	Returns an integer that identifies a particular CPU. The value 128 identifies an AXP system.

2 Adapting Applications to a Larger Page Size

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

2.1 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- or mid-level programming languages. However, your application may 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. The Alpha AXP architecture supports an 8K, 16K, 32K, or 64K byte page size, depending on the implementation, so you should examine the values you specify as arguments to the routines to make sure they still satisfy the requirements of your application. 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.

2.1.1 Compatibility Features

Wherever possible, system services or run-time library routines attempt to present the same interface and return values on AXP systems as they do on VAX systems. For example, on AXP 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 expected by your application are still measured in 512-byte units.

Note

On AXP systems, when creating page frame sections using the $CRMPSC system service (with the SEC$M_PFNMAP flag bit set), the value specified in the page count argument (**pagcnt**) is interpreted as the CPU-specific page size, *not* as a pagelet value.

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

Despite the compatibility, some routines behave differently on AXP systems than they do on VAX systems and may require you to modify your source code. For example, on AXP 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 AXP systems, the routines do not round these addresses to CPU-specific page boundaries.

Table 2–1 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 AXP 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 *OpenVMS System Services Reference Manual*.

Table 2–1 Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on AXP 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 2–1 (Cont.) Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on AXP 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$M_EXPREG flag is set); no rounding is done. (See Section 2.3 for more information about mapping.)
	retadr specifies the actual start- and end-addresses of the memory affected by the call.	Returns the start- and end-addresses of the *usable* range of addresses, which may be different than the total amount mapped. This argument is required when the **relpag** argument is specified.
	flags specifies the type and characteristics of the section to be created or mapped.	The flag bit SEC$M_NO_OVERMAP indicates that existing address space should not be overmapped. When the flag bit SEC$M_PFNMAP is set, the **pagcnt** argument is interpreted as CPU-specific pages, *not* pagelets.
	relpag specifies the page offset at which mapping of the section file should begin.	Interpreted as an index into the section file, measured in pagelets.
	pagcnt specifies the number of pages (blocks) in the file to be mapped.	Interpreted in pagelets; no rounding is done. When the flag bit SEC$M_PFNMAP is set, the **pagcnt** argument is interpreted as CPU-specific pages, *not* pagelets.
	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 AXP systems support 8K, 16K, 32K, and 64K byte physical page sizes, at least 16 pagelets will be 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.

(continued on next page)

Table 2–1 (Cont.) Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on AXP Systems
Expand Program/Control Region ($EXPREG)	**pagcnt** specifies the amount of memory to allocate, in 512-byte units.	Interpreted in pagelets.
	retadr specifies the actual start- and end-addresses of the memory affected by the call.	Unchanged.
Get Job/Process Information ($GETJPI)	**itmlst** specifies which information about the process is to be returned.	Many items, such as JPI$_WSEXTENT, interpreted as pagelet values. See the *OpenVMS System Services Reference Manual* for more information.
Get Queue Information ($GETQUI)	**itmlst** specifies information to be used in performing the function specified by the **func** argument.	Several items interpreted as pagelet values. See the *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 *OpenVMS System Services Reference Manual* for more information.
Get User Authorization Information ($GETUAI)	**itmlst** specifies which information from the user's user authorization file is to be returned.	Several items return pagelet values. See the *OpenVMS System Services Reference Manual* for more information.
Lock Page ($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 Working Set ($LKWSET)	**inadr** specifies the start- and end-addresses of the memory to be locked.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was locked.	Unchanged.
Map Global Section ($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 2.3 for more information about mapping.)

(continued on next page)

Table 2–1 (Cont.) Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on AXP Systems
	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 ($PURGWS)	**inadr** specifies the start- and end-addresses of the memory to be purged.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
Set Protection ($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 ($SETUAI)	**itmlst** specifies which information from the user's user authorization file is to be set.	Several items interpreted in pagelet values. See the *OpenVMS System Services Reference Manual* for more information.
Send to Job Controller ($SNDJBC)	**itmlst** specifies information to be used in performing the function specified by the **func** argument.	Several items interpreted in pagelet values. See the *OpenVMS System Services Reference Manual* for more information.
Unlock Page ($ULKPAG)	**inadr** specifies the start- and end-addresses of the memory to be unlocked.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was unlocked.	Unchanged.
Unlock Working Set ($ULWSET)	**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.
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 will be 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 2–2 allocate (or free) pages of memory. For compatibility, these routines also interpret the page-count information you specify in pagelets.

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

Routine	Argument	Behavior on AXP 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.

2.2 Examining Memory Allocation Routines

To determine if 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

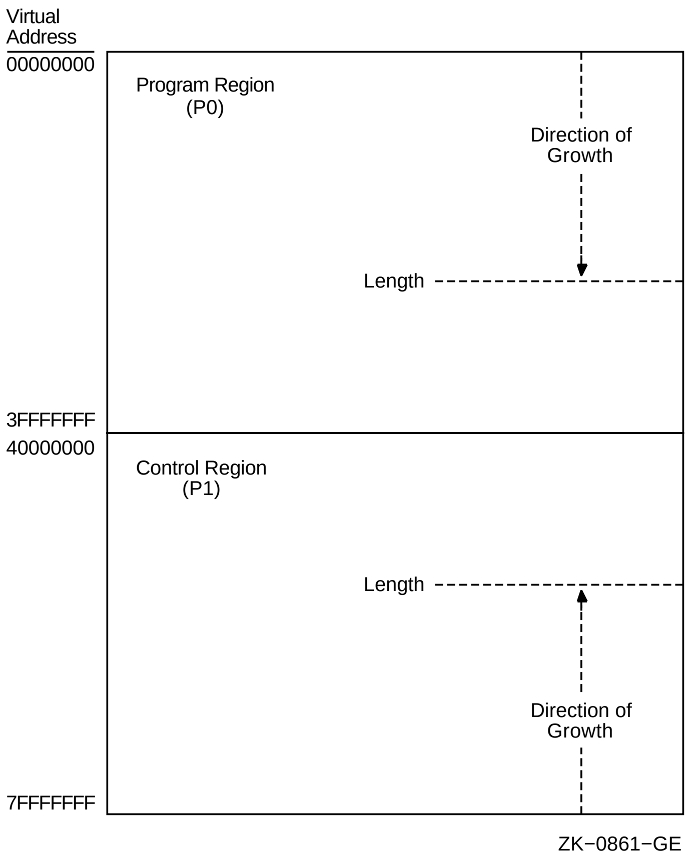
The Alpha AXP architecture defines the same virtual address space layout as the VAX architecture and allows for growth of the P0 and P1 regions in the same direction as on VAX systems. Figure 2–1 illustrates this layout.

2.2.1 Allocating Memory in Expanded Virtual Address Space

If your application allocates memory by *expanding* virtual address space using the $EXPREG system service, you may not need to make any source code changes because the values you specified as arguments are valid on AXP systems and VAX systems. The reasons for this are as follows:

- On AXP 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 may be larger on an AXP 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 2–1 Virtual Address Layout



Recommendation

Your application may not need to be modified. However, Digital 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 may vary among implementations of the Alpha AXP 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 2–1 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 AXP system, the $EXPREG system service allocates at least 8192 bytes and possibly more, depending on the page size of the particular implementation of the Alpha AXP architecture.

Example 2–1 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  1
#define  P0_SPACE   0
#define  P1_SPACE   1

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

2    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 items in the following list correspond to the numbered items in Example 2–1:

1 The example defines a symbol, PAGE_COUNT, to stand for the number of pages requested.

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

2.2.2 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 may 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 AXP systems, the $CRETVA system service rounds the start-address down to a CPU-specific page boundary, which will vary with different implementations.
- The size of the reallocation, specified by the address range in the **inadr** argument, may be larger on an AXP 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, Digital 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 2–2 shows how memory allocated to a buffer can be reallocated by using the $CRETVA system service.

Example 2–2 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);
     }
}
```

2.2.3 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 will vary with the implementation, your application should not make any assumptions regarding the extent of the allocation.

2.3 Examining Memory Mapping Routines

To determine if 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 may be in existing virtual address space)
- Map memory into an existing area of your virtual address space, defined by the start- and end-addresses 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- and end-addresses, 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 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. (Note that this mode of operation of the $CRMPSC system service is not supported on AXP systems. If your application uses this mode, see Section 2.3.2 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 miminum 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.

2.3.1 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 may 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 AXP 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 AXP system.

Recommendation

While applications that map sections into expanded areas of virtual memory may work correctly without modification, Digital 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 2.3.4.

Example 2–3 illustrates 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 CREATE command, as follows:

```
$  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 2–3 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 2–3 (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 );
     }

}
```

2.3.2 Mapping a Single Page to a Specific Location

If your application maps a section file into a single page of memory, you will need to modify your source code because this mode of operation is not supported on AXP systems. Because the page size on AXP systems differs from that on VAX systems and varies with different implementations of the Alpha AXP 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 if your application uses this mode, check the start- and end-addresses 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

Digital suggests the following guidelines when 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 2.3.1.
- If the location into which the mapping occurs is important, define both the start- and end-addresses in the **inadr** argument and map the section into a defined area. For more information about this mode, see Section 2.3.3.

2.3.3 Mapping into a Defined Address Range

If your application maps a section into a defined area of its virtual address space, you may need to modify your source code because, on AXP systems, the $CRMPSC and $MGBLSC system services interpret some of the arguments differently 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 for you. On AXP systems, automatic rounding is not done because rounding to CPU-specific page boundaries affects a much larger portion of memory due to the larger page sizes on AXP systems. Thus, on AXP 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 AXP systems as it is on VAX systems. See Section 2.3.4 for more information.

Recommendations

Digital 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, Digital recommends the following guidelines:

- Because the operating system maps a minimum of one physical page and physical pages on AXP 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 AXP systems, you would need to declare a buffer in your application as large as the largest possible AXP page, 64K 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 will not overwrite neighboring data because the linker allocates image sections on page boundaries; neighboring data will start 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 *OpenVMS Linker Utility Manual*.) Note that you may need to use the capabilities of whatever high- 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- and end-addresses of the section that you specify as arguments to the $CRMPSCK and $MGBLSC system services are aligned with the start- and end-addresses of a CPU-specific page. On VAX systems, the system services round the addresses to page boundaries for you. On AXP 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 AXP systems, if your application specifies the **relpag** argument to the $CRMPSC system service, you *must* specify the **retadr** argument.)

For example, the VAX program in Example 2–4 maps the section file created in Section 2.3.1 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 2–4 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 2–4 (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 );
    }

}
```

To get the program in Example 2–4 to run correctly on an AXP 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 AXP page boundary and the end-address specified is aligned with the end of an AXP page.
- You must ensure that when a larger page on an AXP 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 is different in various programming languages on each platform. Check 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 it 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 2.4.) Note that you do not need to change the allocation of the buffer into which the section is mapped.

To ensure that your application will run on an AXP system with any page size, specify the /BPAGE=16 qualifier to force the linker to align image sections on 64K-byte boundaries. Note that the total amount of memory mapped may 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 2–5 shows the source changes required for Example 2–4 to get it to run on an AXP system.

Example 2–5 Source Code Changes Required to Run Example 2–4 on an AXP 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> 1

char buffer[512];  2
char *filename = "maptest.dat";
struct FAB fab;

long  cpu_pagesize; 3

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]; 4

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 2–5 (Cont.) Source Code Changes Required to Run Example 2–4 on an AXP 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;

5    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]; 6

     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 items in the following list correspond to the numbered items in Example 2–5:

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

2 The buffer is defined without using the __align(page) storage descriptor. Because the page size cannot be determined until run time on OpenVMS AXP systems, the DEC C for OpenVMS AXP compiler aligns the data on the largest AXP page size (64K bytes) when __align(page) is specified.

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

4 This variable will hold the page-size value returned.

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

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

2.3.4 Mapping from an Offset into a Section File

Your application may 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 interpets the value of the **relpag** argument in 512-byte units on both VAX systems and AXP systems. Note, however, that because the CPU-specific page size on AXP 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 AXP systems, the mapping of the section file will start at the beginning of the CPU-specific page that contains the offset address, *not* at the address specified by the offset.

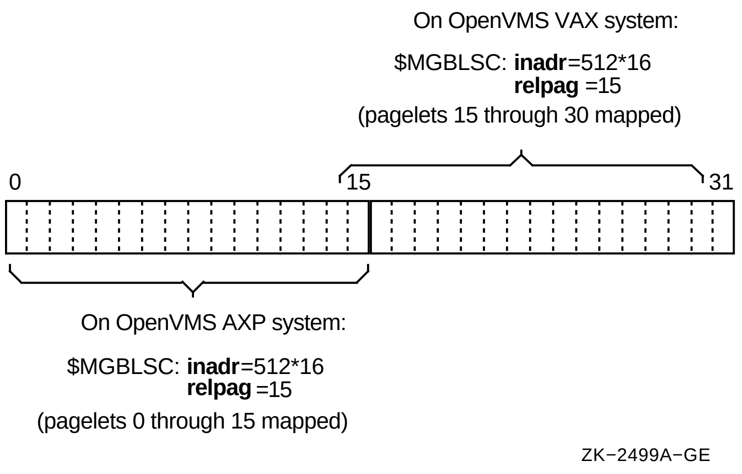
Note

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 may 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 AXP systems. If the address range specified is too small, your application may not map the entire portion of the section file you desire, because the mapping begins at an earlier starting 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 AXP system, you must allow for the difference in page sizes. For example, on an AXP system with an 8K-byte page size, the address specified by the **relpag** offset might fall 15 pagelets into a CPU-specific page, as illustrated in Figure 2–2. Because the $CRMPSC system service on an AXP system begins the 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 would need to 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 AXP system. Otherwise, only one block of the portion of the section file you specified would be mapped. Figure 2–2 illustrates this scenario.

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


On OpenVMS VAX system:
$MGBLSC: **inadr**=512*16
relpag =15
(pagelets 15 through 30 mapped)
0
15
31
On OpenVMS AXP system:
$MGBLSC: **inadr**=512*16
relpag =15
(pagelets 0 through 15 mapped)
ZK−2499A−GE

When trying to calculate how much to enlarge the size of the address range specified in the **relpag** argument, the following formula may 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.

2.4 Obtaining the Page Size at Run Time

To obtain the page size supported by an AXP system, use the $GETSYI system service. Example 2–6 illustrates how to use this system service to obtain the page size at run time.

Example 2–6 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 2–6 (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 );
     }
}
```

2.5 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 both VAX and AXP systems. The system service rounds the addresses to CPU-specific page boundaries if necessary.

However, because Alpha AXP instructions cannot contain full virtual addresses, AXP images must reference procedures and data indirectly through a pointer to a procedure descriptor. The procedure descriptor contains information about the procedure, including the actual code address. These pointers to procedure descriptors and data are collected into a new program section called a **linkage section**.

Recommendation

On AXP systems, it is not sufficient to simply lock a section of code into memory to improve performance. You must also lock the associated linkage section into the working set.

To lock the linkage section in memory, determine the start- and end-addresses of the linkage section and pass these addresses as values in the **inadr** argument to a call to the $LKWSET system service.

3
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.

3.1 Overview

If your application uses multiple threads of execution and the threads share access to data, you may need to add explicit synchronization mechanisms to your application to protect the integrity of the shared data on AXP systems. Without synchronization, an access to the data initiated by one application thread can potentially 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 either be initiated 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 may not explicitly protect the shared data. Instead, these applications may 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 3.1.1).

For example, applications that use a semaphore variable to synchronize access to a critical region of code by multiple threads depend on the semaphore being incremented atomically. On VAX systems, this is guaranteed by the VAX architecture. The Alpha AXP architecture does not make the same synchronization guarantees. On AXP systems, access to this semaphore or any data that can be accessed by multiple threads of execution must be explicitly synchronized. Section 3.1.2 describes features of the Alpha AXP architecture you can use to provide equivalent protection.

3.1.1 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. (Note that 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, noninterruptable sequence (called an **atomic** operation) from the viewpoint of multiple application threads executing on a single processor. The Alpha AXP architecture does not support such instructions. Operations that could be performed atomically on VAX systems require a sequence of instructions on AXP systems, which can be interrupted, leaving the data in an unpredictable state.

  For example, the VAX Increment Long (INCL) instruction fetches the contents of a specified longword, increments its value, and stores the value back in the longword, performing the operations without interruption. On AXP systems, each step must be explicitly performed by a separate instruction.

  To provide compatibility with VAX systems, the Alpha AXP architecture defines a pair of instructions that you can use to ensure that a read/write operation is done atomically. Section 3.1.2 describes these instructions and describes how compilers on AXP systems make this capability available to programs written in high-level languages.

  Note, however, that 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 implement certain program statements, such as an increment operation (`X = X + 1`), using a VAX atomic instruction, even if such an instruction is available.

- **Memory access granularity**—The VAX architecture supports instructions that can manipulate byte- and word-sized data in a single, noninterruptable operation. (The VAX architecture supports instructions to manipulate data of other sizes as well.) The Alpha AXP architecture supports instructions that manipulate longword- and quadword-sized data. Manipulation of byte- and word-sized data on AXP systems requires multiple instructions: the longword or quadword that contains the byte or word must be fetched, the nontargeted bytes must be masked, the target byte or word manipulated, and then the entire longword or quadword must be stored. Because this sequence is interruptable, operations on byte and word data, which are atomic on VAX systems, are not atomic on AXP systems.

  Note that this change in the granularity of memory access can also affect the definition of which data is shared. On VAX systems, a byte- or word-sized data item that is shared can be manipulated individually. On AXP systems, the entire longword or quadword that contains the byte- or word-sized item must be manipulated. Thus, simply because of its proximity to an explicitly shared data item, neighboring data may become *unintentionally* shared.

  Compilers use the Alpha AXP instructions described in Section 3.1.2 to ensure the integrity of byte- 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. AXP 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 visible to threads executing concurrently on an AXP multiprocessor system require explicit synchronization.

  To provide compatibility with VAX systems, the Alpha AXP architecture supports an instruction with which you can ensure that read/write operations occur in the order specified, from the viewpoint of all the processors in the system. Section 3.1.2 provides more information about this instruction and about how high-level languages make this instruction available. Section 3.3 describes the feature of the Alpha AXP architecture that provides this synchronization and describes how the compilers make it available to high-level language programs on AXP systems.

3.1.2 Alpha AXP Compatibility Features

To provide compatibility with the atomicity capabilities of the VAX architecture, the Alpha AXP architecture defines two mechanisms:

- **Load-locked/Store-conditional instructions**—The Alpha AXP instruction set includes a pair of instructions, named Load-locked (LD*x*L) and Store-conditional (ST*x*C), that provide for atomic load and store operations by setting and testing a lock bit. For complete information about these instructions, see the *Alpha Architecture Reference Manual*.

  Using the Load-locked/Store-conditional instructions, compilers can provide atomic access to byte- and word-sized data on AXP systems. In addition, compilers may generate the Load-locked/Store-conditional instruction sequence when accessing byte- and word-sized data that is declared with the **volatile** attribute. (The Alpha AXP architecture provides atomic load and store operations of longword- and quadword-sized data.)

- **Memory barriers**—The Alpha AXP 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 barrier (MB), guarantees that all subsequent load or store instructions will not access memory until after all previous load and store instructions have accessed memory from the viewpoint of multiple threads of execution.

3.2 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. Unintentionally shared data is shared 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.

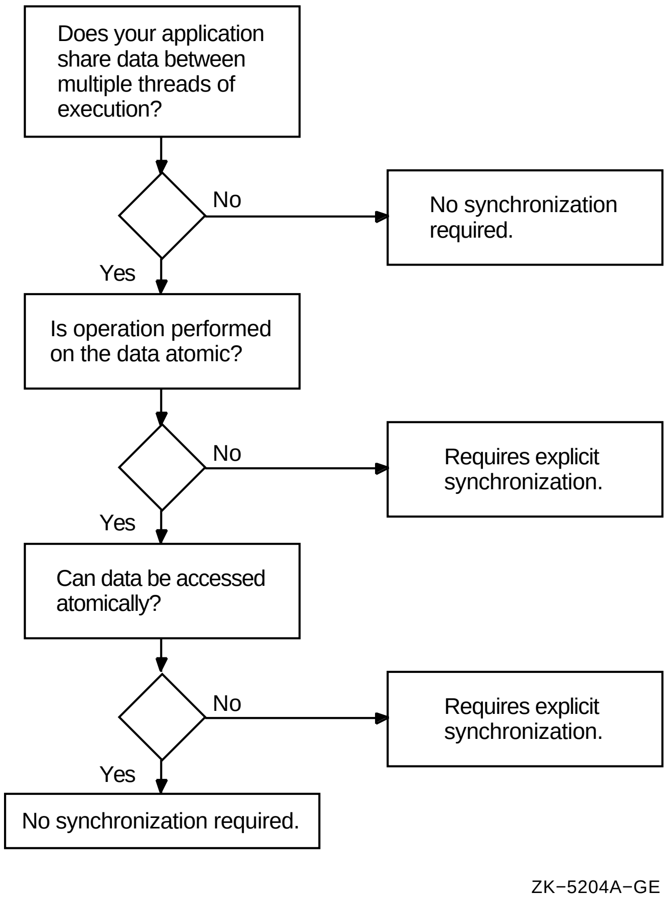
Because compilers on AXP systems use quadword instructions by default in certain circumstances, all data items within a quadword of a shared data item may potentially become unintentionally shared. For example, compilers use quadword instructions to access a data item that is not aligned on natural boundaries. (Data is naturally aligned when its address is divisible by its size. For more information, see Chapter 4. Compilers align explicitly declared data on natural boundaries by default.)

When examining data access, determine if another thread could view the data in an intermediate state and, if such a view is possible, whether it is important to the application. In some cases, the exact value of the shared data may 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 3–1 illustrates this decision process.

Figure 3–1 Synchronization Decision Tree


Does your application share data between multiple threads of execution?
No
No synchronization required.
Yes
Is operation performed on the data atomic?
No
Requires explicit synchronization.
Yes
Can data be accessed atomically?
No
Requires explicit synchronization.
Yes
No synchronization required.
ZK−5204A−GE

3.2.1 Protecting Explicitly Shared Data

The program in Example 3–1 is a simplified illustration of some possible atomicity assumptions in a VAX application. The program uses a variable, *flag*, through which an AST thread communicates with a main processing thread of execution. In the example, 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.

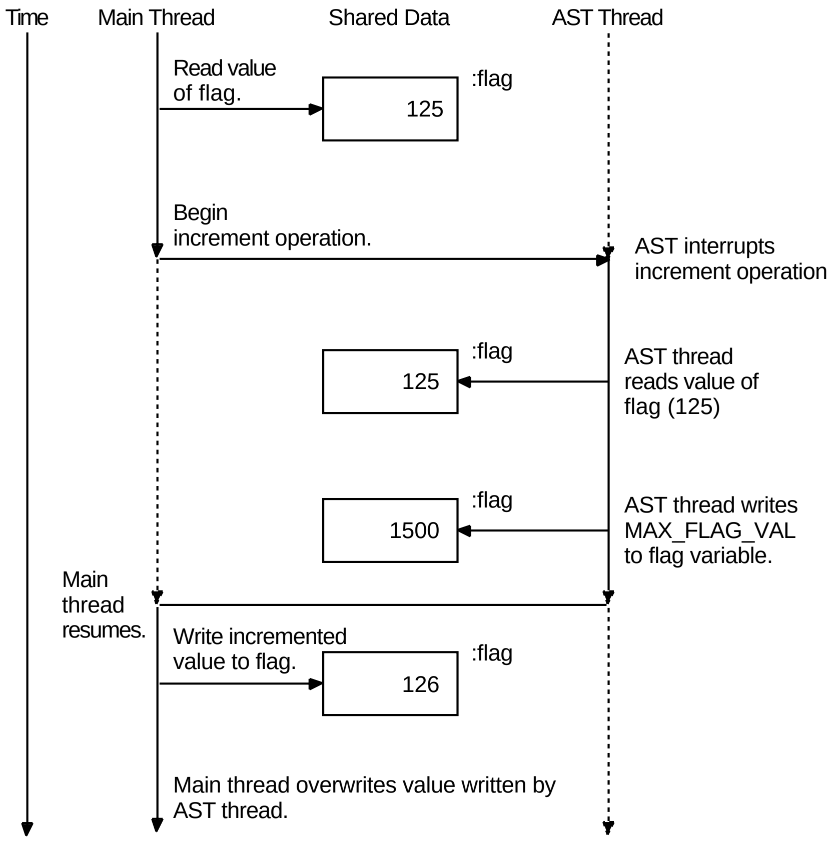
Example 3–1 Atomicity Assumptions in a Program with an AST Thread

```
#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 */

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 3–1, the variable named *flag* 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 an AXP system, this program may not always work as desired 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 illustrated in Figure 3–2. (This would be more likely to fail in a real application with dozens of AST threads.) In this scenario, the AST thread would interrupt 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 would complete, overwriting the value of the AST thread. When the loop test is performed, the value would not be at its maximum and the processing loop would continue.

Figure 3–2 Atomicity Assumptions in Example 3–1


Time
Main Thread
Shared Data
AST Thread
Read value of flag.
:flag
125
Begin increment operation.
AST interrupts increment operation
:flag
125
AST thread reads value of flag (125)
:flag
1500
AST thread writes MAX_FLAG_VAL to flag variable.
Main thread resumes.
Write incremented value to flag.
:flag
126
Main thread overwrites value written by AST thread.

ZK−5203A−GE

Recommendations

To correct these atomicity dependencies, Digital 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, DEC C for OpenVMS AXP 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 3–1, replace the increment operation, which is performed by the C increment operator (flag++) with the atomicity built-in supported by DEC C for OpenVMS AXP systems (__ADD_ATOMIC_LONG(&flag,1,0)). See Example 3–2 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- or word-sized data to a longword or quadword, change the granularity the compiler uses when accessing the data item. Many compilers on AXP systems allow you to specify the granularity they will use when accessing a particular data item or when processing an entire module. Note, however, that specifying byte and word granularity can have an adverse effect on the performance of your application.

Example 3–2 illustrates how these changes are implemented in the program presented in Example 3–1.

Example 3–2 Version of Example 3–1 with Synchronization Assumptions

```
#include <ssdef.h>
#include <descrip.h>
#include <builtins.h> 1

#define MAX_FLAG_VAL  1500
int    ast_rout();
long  time_val[2];
int 2       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 );
     }
```

(continued on next page)

Example 3–2 (Cont.) Version of Example 3–1 with Synchronization Assumptions

```
     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); 3
     }
     printf("Done\n");
}
ast_rout()     /*  sets flag to maximum value to stop processing */
{
      flag = MAX_FLAG_VAL;
}
```

The items in the following list correspond to the numbers in Example 3–2:

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

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

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

3.2.2 Protecting Unintentionally Shared Data

In Example 3–1, both threads clearly access the same variable. However, on an AXP system, it is possible for an application to 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 AXP 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 4.)

To illustrate this problem, consider a modified version of the program in Example 3–1 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 would be unpredictable, depending on the timing of the two operations.

Recommendations

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

- Change the size of the shared variables to longwords or quadwords. Note, however, that because compilers on AXP 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. The compilers align data on natural boundaries by default on AXP systems.

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 DEC 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 employing interlocked queue operations.

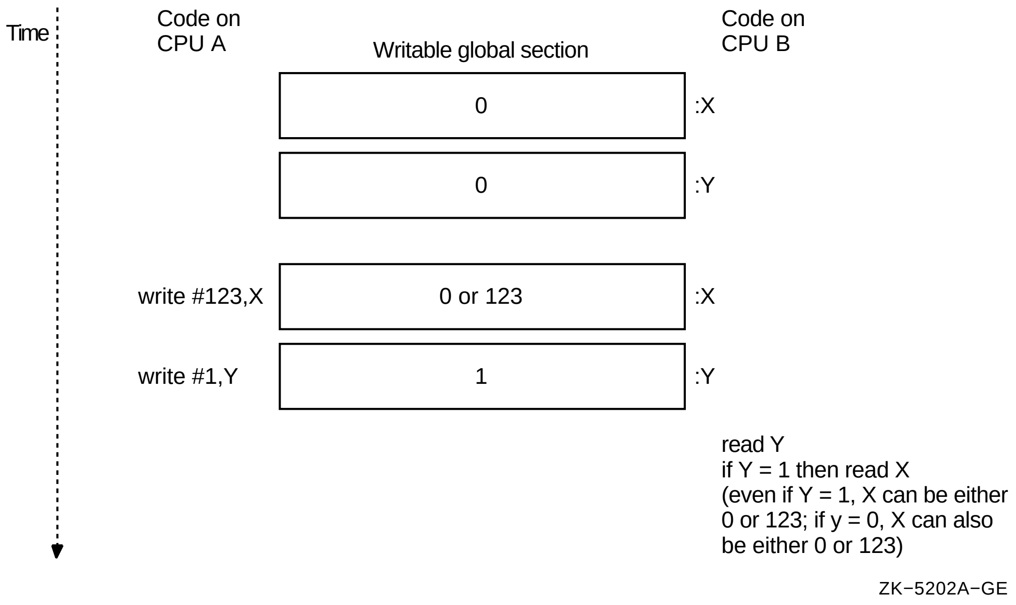
3.3 Synchronizing Read/Write Operations

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 3–3) and then writes a flag (represented by Y in Figure 3–3), CPU B can determine that the data buffer has changed by examining the value of the flag.

On AXP systems, read and write operations to memory may 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 may become visible to CPU B in an order different from that in which they were written.

Figure 3–3 illustrates 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 may see the flag bit set *before* the data is actually written and erroneously use stale memory contents.

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



Recommendations

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

- Use the Alpha AXP memory barrier instruction (MB) before and after all read and write instructions for which the completion order is crucial. For example, the DEC C for OpenVMS AXP systems compiler supports the memory barrier instruction as a built-in function.
- Redesign the application to use the memory interlocks available in the PPL$ run-time library 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.

3.4 Ensuring Atomicity in Translated Images

The VEST command's /PRESERVE qualifier accepts keywords that allow translated VAX images to run on AXP 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 have an adverse effect on the performance of your application. (For complete information about specifying the /PRESERVE qualifier, see *DECmigrate for OpenVMS AXP 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.

4 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 AXP system.

4.1 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 may be able to successfully recompile and run an application that runs on VAX systems on an AXP system without modifying the data declarations it contains.

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

- **Assumptions about data-type mappings**—Your application may depend on the underlying VAX data type to which a high-level language maps. The Alpha AXP architecture supports most of the VAX data types; however, there are some data types that are not supported. Your application may make assumptions about the size or bit format of a data type that may no longer be valid on an AXP system. Section 4.2 provides more information about this topic.
- **Assumptions about data-type selection**—Your choice of data type may have different implications on an AXP system. For example, on VAX systems, you may have chosen the smallest data type available to represent data items to conserve memory usage. On an AXP system, this strategy may actually increase memory usage. Section 4.3 provides more information about this topic.

4.2 Checking for Dependence on a VAX Data Type

To provide data compatibility, the Alpha AXP architecture has been designed to support many of the same native data types as the VAX architecture. Table 4–1 lists the native data types supported by both architectures. (See the *Alpha Architecture Reference Manual* for more information about the formats of the data types.)

Table 4–1 Comparison of VAX and AXP Native Data Types

VAX Data Types	AXP Data Types
byte	byte
word	word
longword	longword
quadword	quadword
octaword	–
F_float	F_float
D_float (56-bit precision)	D_float (53-bit precision)
G_float	G_float
H_float	–
–	S_float (IEEE)
–	T_float (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	–

Recommendations

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

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

- **D_float data**—Most compilers on AXP systems map their double-precision floating-point data type to the VAX native G_float data type by default because the Alpha AXP architecture does not support the VAX D_float data type. The OpenVMS VAX compilers map their double-precision floating-point data type to the D_float data type. For example, VAX C maps the double data type to D_float and DEC C for OpenVMS AXP systems compiler maps the double data type to the G_float data type.

  This change may not affect most applications. Note, however, that the value returned by the G_float data type (significant to 15 digits after the decimal) is slightly less precise than the value returned by the D_float data type (significant to 16 digits after the decimal).

The OpenVMS Run-Time Library supports a conversion routine (CVT$CONVERT_FLOAT) that you can use to convert floating-point data from one format to another. For example, using this routine you can convert data in D_float format to IEEE format and back again. Note also that the Alpha AXP architecture supports the IEEE double-precision floating-point format (T_float).

DEC C for OpenVMS AXP systems issues a warning message when it encounters declarations that use the long float data type. On VAX systems, the long float data type is a synonym for double. On AXP systems, the long float data type is obsolete, even when the DEC C compiler is used in VAX C mode.

- **Pointer data**—Check for assumptions that an address (pointer) data type is equivalent in size to an integer data type. On AXP systems, an address is 64 bits.

  For example, in VAX C, some programs may make this assumption, as illustrated in Example 4–1.

Example 4–1 Assumptions About Data Types in VAX C Code

```
typedef struct {
       char     small;
       short   medium;
       long     large;
       } MYSTRUCT ;

main()
{

    int        a1;
    long       b1;
    MYSTRUCT   c1;

1   a1 = &c1;
2   b1 = &c1;

3   a1->small = 1;
    b1->small = 2;

}
```

The items in the following list correspond to the numbered items in Example 4–1:

1 The example assigns the address of the structure to the variable *a1*, declared as an int data type.

2 The example assigns the address of the structure to the variable *b1*, declared as a long data type.

3 The example accesses the first field in the structure by using the variables assigned to int and long data types.

To move this example to an AXP system, you should change the declarations of *a1* and *b1* to be pointers to the data structure (MYSTRUCT), as in the following:

```
MYSTRUCT *a1,*b2;
```

4.3 Examining Assumptions About Data-Type Selection

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

4.3.1 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.

On an AXP system, byte- and word-sized data incurs more overhead than longword- or quadword-sized data because the Alpha AXP architecture does not support instructions that manipulate these smaller data types. Each reference to a byte or word, which generates a single instruction on a VAX system, generates a sequence of instructions on an AXP system, in which the longword containing the byte or word is fetched, manipulated so that only the target bytes are modified, and then stored. For frequently referenced data, these additional instructions can significantly add to the total size of your application on an AXP system.

4.3.2 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.)

Accessing unaligned data incurs more overhead that accessing aligned data on both VAX and AXP systems. However, VAX systems use microcode to minimize the performance impact of unaligned data. On AXP systems, there is no hardware assistance. References to unaligned data trigger a fault, which must be handled by the operating system's unaligned fault handler. While the fault is being handled, the instruction pipeline must be stopped. Thus, the cost of an unaligned reference in performance is dramatically higher on AXP systems.

The compilers on AXP 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 a run-time unaligned fault from occurring. 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- 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- or word-sized data item is frequently referenced, changing it to a longword eliminates the extra instructions required 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 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 may 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- 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.
- For static data, always use a longword instead of a byte. It does incur three extra bytes of storage; however, a single reference requires three extra instructions, each of which is a longword.
- Use the capabilities of the compilers on AXP systems to uncover data that is not aligned on natural boundaries. For example, many compilers on AXP systems 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 *OpenVMS Debugger Manual*.
- Take advantage of the natural alignment provided by the compilers on AXP systems, wherever interoperability concerns allow. On AXP systems, compilers align data on natural boundaries by default, wherever possible. On VAX systems, compilers use byte alignment.

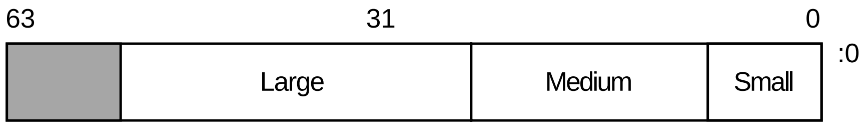
  Note that the compilers on AXP systems support qualifiers and language pragmas that allow you to request they use the same byte alignment they use on VAX systems. For example, the DEC C for OpenVMS AXP systems compiler supports the /NOMEMBER_ALIGNMENT qualifier and a corresponding pragma that allow you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

The data structure defined in Example 4–1 illustrates these data-type selection concerns. The structure definition, called mystruct, is made up of byte-, word-, and longword-sized data, as follows:

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

When compiled using VAX C, the structure is laid out in memory as illustrated in Figure 4–1.

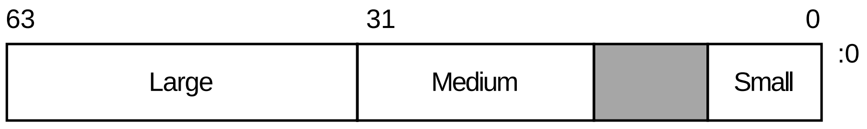
Figure 4–1 Alignment of mystruct Using VAX C



ZK−5209A−GE

When compiled using DEC C for OpenVMS AXP systems compiler, the structure is padded to achieve natural alignment, as illustrated in Figure 4–2. Note that by adding a byte of padding after the first field, small, both the following members of the structure are aligned.

Figure 4–2 Alignment of mystruct Using DEC C for OpenVMS AXP Systems



ZK−5210A−GE

Note that the byte- 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. If, however, 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.

5 Examining the Condition Handling Code in Your Application

This chapter describes the effect of differences between the VAX architecture and the Alpha AXP architecture on the condition handling code in your application.

5.1 Overview

For the most part, the condition handling code in your application will work correctly on an AXP system, 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 Alpha AXP condition handling facility and the VAX condition handling facility that may require you to modify your source code. These include the following:

- 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.

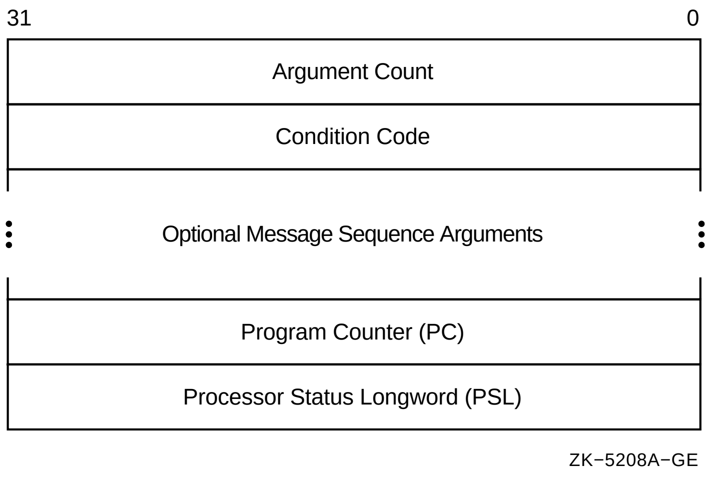
The following sections describe these changes in more detail and provide guidelines to help you decide if modifying your source code is necessary.

5.2 Examining Condition Handling Routines for Dependencies

The calling sequence of user-written condition handling routines remains the same on AXP systems as it is 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 *OpenVMS Programming Concepts Manual*. On AXP systems, the data returned in the signal array and its format is the same as it is on VAX systems, as illustrated in Figure 5–1.

Figure 5–1 Signal Array on VAX and AXP Systems

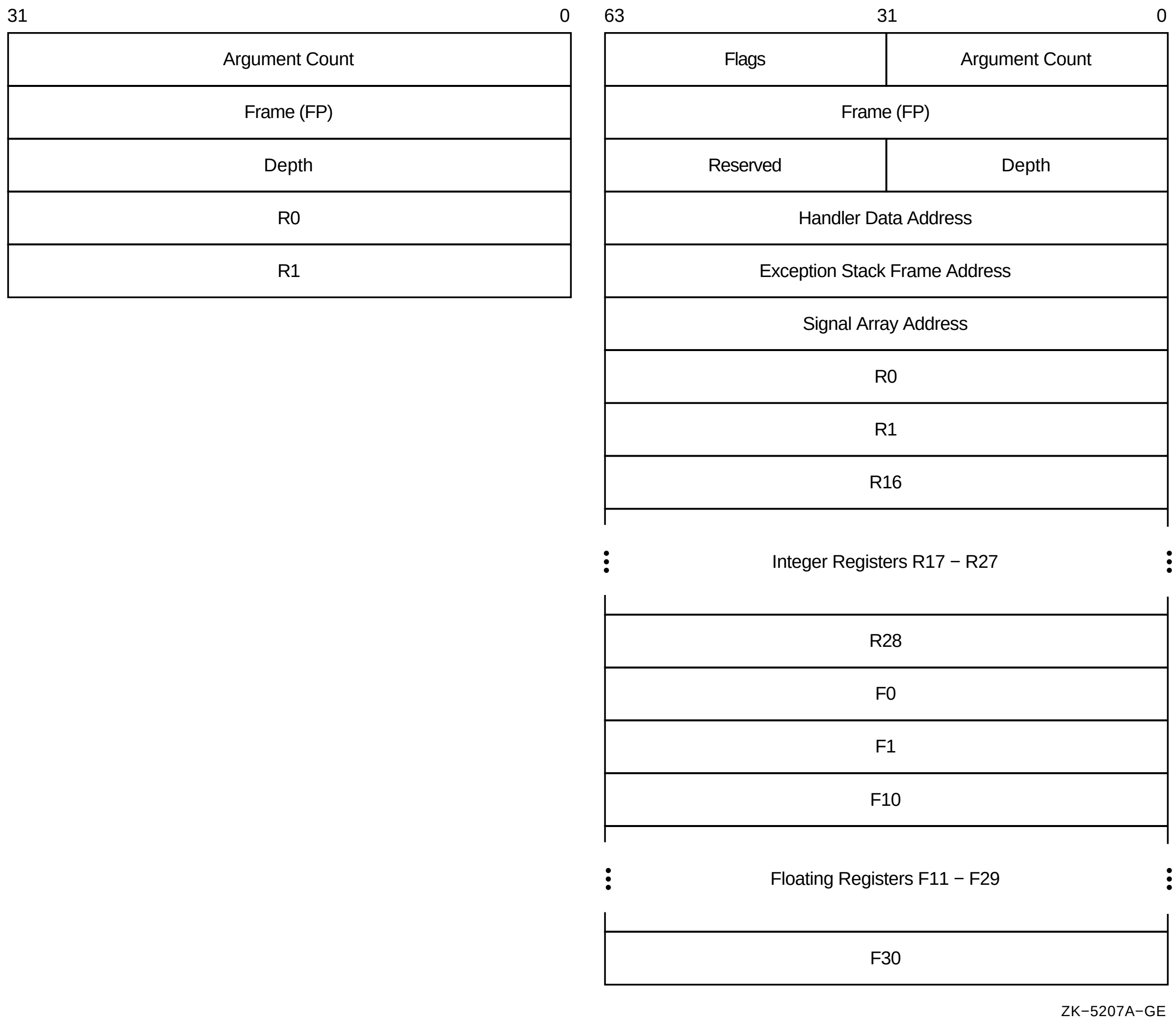

31
0
Argument Count
Condition Code
Optional Message Sequence Arguments
Program Counter (PC)
Processor Status Longword (PSL)
ZK−5208A−GE

The following table describes the arguments in the signal array:

Argument	Description
Argument Count	On AXP and VAX systems, this argument contains a positive integer that indicates the number of longwords that follow in the array.
Condition Code	On AXP 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 AXP systems, is described in *OpenVMS Programming Interfaces: Calling a System Routine*. Note, however, that AXP systems do not support every condition code returned on VAX systems and defines condition codes that cannot be returned on a VAX system. Section 5.3 lists VAX condition codes that cannot be returned on AXP systems.
Optional Message Sequence	These arguments provide additional information about the particular exception returned and vary for each exception. The *OpenVMS Programming Concepts Manual* describes these arguments for VAX exceptions.
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 AXP systems, this argument contains the lower 32 bits of the PC (which is 64 bits long on AXP systems).
Processor Status Longword (PSL)	A formatted 32-bit argument that describes the status of the processor when the exception occurred. On AXP systems, this argument contains the lower 32 bits of the Alpha AXP 64-bit processor status (PS) quadword.

On AXP 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 AXP 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 AXP systems, not longwords (32 bits) as it is on VAX systems. Figure 5–2 compares the format of the mechanism array on VAX and AXP systems.

Figure 5–2 Mechanism Array on VAX and AXP Systems



The following table describes the arguments in the mechanism array:

Argument	Description
Argument Count	On VAX systems, this argument contains a positive integer that represents the number of longwords that follow in the array. On AXP systems, this argument represents the number of *quadwords* in the mechanism array, not counting the argument count quadword (always 43 on AXP systems).
Flags	On AXP systems, this argument contains various flags to communicate additional information. For example, if bit 0 is set, it indicates that the process has already performed a floating-point operation and the floating-point registers in the array are valid. (No equivalent in the mechanism array on VAX systems.)

Argument	Description
Frame Pointer (FP)	On VAX and AXP systems, this argument contains the address of the call frame on the stack that established the condition handler.
Depth	On VAX and AXP systems, this argument 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 AXP systems, this argument 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 AXP systems, this argument contains the address of the exception stack frame. (No equivalent in the mechanism array on VAX systems.)
Signal Array Address	On AXP systems, this argument contains the address of the signal array. (No equivalent in the mechanism array on VAX systems.)
Registers	On VAX and AXP systems, the mechanism array includes the contents of scratch registers. On AXP systems, this includes a much larger set of registers and also includes a corresponding set of floating-point registers.

Recommendations

Because the signal array is the same on AXP systems as it is on VAX systems, you may not need to modify the source code of your condition handling routine. However, the changes to the mechanism array may require changes to your source code. In particular, check 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 may need to modify your source code. Because of architectural changes, the depth argument returned on an AXP system may be different from that returned on a VAX system. (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$UNWIND system service, or unwind to the caller of the establisher frame by using the default depth argument of the SYS$UNWIND system service, will continue to work correctly. Depths specified as negative numbers still indicate exception vectors (as on VAX systems).

Example 5–1 presents a condition handling routine written in C.

Example 5–1 Condition Handling Routine

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

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

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

The items in the following list correspond to the numbered items in Example 5–1:

1. 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.
2. This condition handling routine uses the LIB$MATCH_COND run-time library routine to compare the returned condition code with the condition code that identifies integer overflow (defined in SSDEF.H). The condition code is referenced as a field in the system-defined signal data structure (defined in CHFDEF.H).
3. The LIB$MATCH_COND routine returns a nonzero result when a match is found. The condition handling routine executes different code paths based on this result.

5.3 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 5–1, illustrates how a condition handling routine can accomplish this task by using the run-time library routine LIB$MATCH_COND:

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

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

For software exceptions, AXP 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, however, are more architecture specific, especially the arithmetic exceptions. Only a subset of the hardware exceptions supported by VAX systems (documented in the *OpenVMS Programming Concepts Manual*) are also supported on AXP systems. In addition, the Alpha AXP architecture defines several additional exceptions that are not supported by the VAX architecture.

Table 5–1 lists the VAX hardware exceptions that are not supported on AXP systems and the Alpha AXP 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 may need to add the code to test for the equivalent Alpha AXP exceptions. (Section 5.3.1 provides more information about testing for arithmetic exceptions on AXP systems.)

Note

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

Table 5–1 Architecture-Specific Hardware Exceptions

Exception Condition Code	Comment
Exceptions Specific to AXP Systems	
SS$_HPARITH–High-performance arithmetic exception	Replaces VAX arithmetic exceptions (see Section 5.3.1)
SS$_ALIGN–Data alignment trap	No equivalent on VAX systems
Exceptions Specific to VAX Systems	
SS$_ARTRES–Reserved arithmetic trap	No equivalent on AXP systems
SS$_COMPAT–Compatibility fault	No equivalent on AXP systems
[1]SS$_DECOVF–Decimal overflow	Replaced by SS$_HPARITH (see Section 5.3.1)
[1]SS$_FLTDIV–Float divide-by-zero (trap)	Replaced by SS$_HPARITH (see Section 5.3.1)
SS$_FLTDIV_F–Float divide-by-zero (fault)	Replaced by SS$_HPARITH (see Section 5.3.1)
[1]SS$_FLTOVF–Float overflow (trap)	Replaced by SS$_HPARITH (see Section 5.3.1)
SS$_FLTOVF_F–Float overflow (fault)	Replaced by SS$_HPARITH (see Section 5.3.1)
[1]SS$_FLTUND–Float underflow (trap)	Replaced by SS$_HPARITH (see Section 5.3.1)

[1]May be generated by software on AXP systems

(continued on next page)

Table 5–1 (Cont.) Architecture-Specific Hardware Exceptions

Exception Condition Code	Comment
Exceptions Specific to VAX Systems	
SS$_FLTUND_F–Float underflow (fault)	Replaced by SS$_HPARITH (see Section 5.3.1)
[1]SS$_INTDIV–Integer divide-by-zero	Replaced by SS$_HPARITH (see Section 5.3.1)
[1]SS$_INTOVF–Integer overflow	Replaced by SS$_HPARITH (see Section 5.3.1)
SS$_TBIT–Trace pending	No equivalent on AXP systems
SS$_OPCCUS–Opcode reserved to customer	No equivalent on AXP systems
SS$_RADMOD–Reserved addressing mode	No equivalent on AXP systems
SS$_SUBRNG–INDEX subscript range check	No equivalent on AXP systems

[1]May be generated by software on AXP systems

5.3.1 Testing for Arithmetic Exceptions on AXP Systems

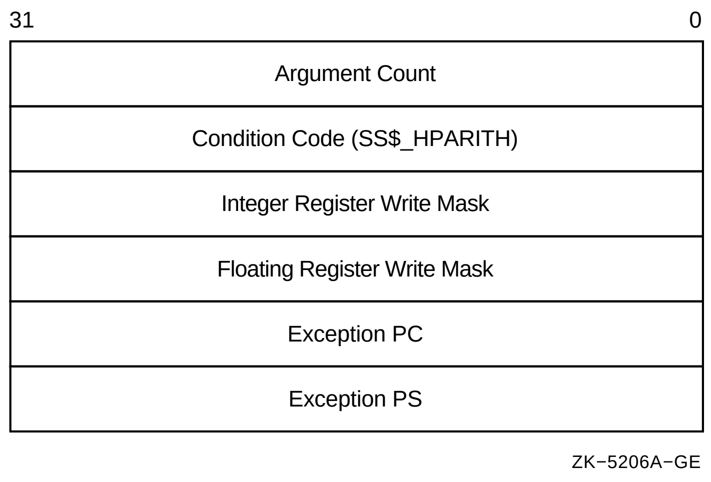
On a VAX system, the architecture ensures that arithmetic exceptions are reported synchronously; that is, a VAX arithmetic instruction that causes an exception (such as an overflow) enters any exception handlers immediately and no subsequent instructions are executed. The program counter (PC) reported to the exception handler is that of the failing arithmetic instruction. This allows application programs, for example, to resume the main sequence, with the failing operation being emulated or replaced by some equivalent or alternate set of operations.

On AXP systems, arithmetic exceptions are reported asynchronously; that is, implementations of the architecture can allow a number of instructions (including branches and jumps) to execute beyond that which caused the exception. These instructions may overwrite the original operands used by the failing instruction, thus causing information integral to interpreting or rectifying the exception to be lost. The PC reported to the exception handler is not that of the failing instruction, but rather is that of some subsequent instruction. When the exception is reported to an application's exception handler, it may be impossible for the handler to fix up the input data and restart the instruction.

Because of this fundamental difference in arithmetic exception reporting, AXP systems define a single condition code, SS$_HPARITH, to indicate all of the arithmetic exceptions. Thus, if your application contains a condition handling routine that performs processing when an integer overflow exception occurs, on VAX systems it expects to receive the SS$_INTOVR condition code. On AXP systems, this exception is indicated by the condition code SS$_HPARITH. In this way, condition handling routines in applications cannot mistake an AXP arithmetic exception with the corresponding VAX exception. This is important because the processing performed by the applications may be architecture specific.

Figure 5–3 illustrates the format of the SS$_HPARITH exception signal array.

Figure 5–3 SS$_HPARITH Exception Signal Array


31
0
Argument Count
Condition Code (SS$_HPARITH)
Integer Register Write Mask
Floating Register Write Mask
Exception PC
Exception PS
ZK−5206A−GE

This signal array contains three arguments that are specific to the SS$_HPARITH exception: the **integer register write mask**, the **floating register write mask**, and the **exception summary** arguments. The integer and floating register mask arguments indicate the registers that were targets of instructions that set bits in the exception summary argument. Each bit in the mask represents a register. The exception summary argument indicates the type of exception (or exceptions) that is being signaled by setting flags in the first seven bits. Table 5–2 lists the meaning of each of these bits when set.

Table 5–2 Exception Summary Argument Fields

Bit	Meaning
0	Software completion.
1	Invalid floating arithmetic, conversion, or comparison operation.
2	Invalid attempt to perform a floating divide operation with a divisor of zero. Note that integer divide-by-zero is not reported.
3	Floating arithmetic or conversion operation overflowed the destination exponent.
4	Floating arithmetic or conversion operation underflowed the destination exponent.
5	Floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.
6	Integer arithmetic or conversion operation from floating point to integer overflowed the destination precision.

Recommendations

The following recommendations provide guidelines for determining if a condition handling routine that performs processing in response to an arithmetic exception needs modification to run on an AXP system:

- 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 does not matter that the exception is delivered asynchronously on AXP systems. These condition handling routines require only the addition of a test for the SS$_HPARITH condition code.

- If your application attempts to restart the operation that caused the exception, you must either rewrite your code or use a compiler qualifier that ensures the exact reporting of arithmetic exceptions. (See Appendix A for more information about these compiler qualifiers.) Note, however, that specifying these instructions can affect performance adversely.
- To guarantee precise reporting of arithmetic exceptions in translated images, specify the /PRESERVE=FLOAT_EXCEPTIONS qualifier on the VEST command line when translating the image. When this qualifier is used, the VEST utility generates code that allows an exception to be reported after each instruction that could result in a floating-point fault. This qualifier adversely affects the performance of the translated image. For more information about using the VEST command, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

Note

A translated VAX image running on an AXP system can return VAX exception conditions, including arithmetic exception conditions.

5.3.2 Testing for Data-Alignment Traps

On an AXP system, 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, without using an Alpha AXP instruction that takes an unaligned address as an operand (LDQ_U). (For more information about data alignment, see Chapter 4.)

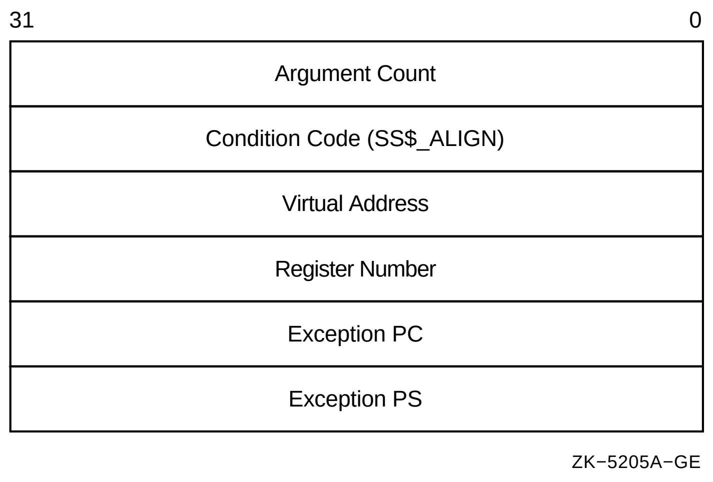
Compilers on AXP systems typically avoid triggering alignment faults by:

- 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, however, that compilers cannot align dynamically defined data. Thus, alignment faults may be triggered.

An alignment exception is identified by the condition code SS$_ALIGN. Figure 5–4 illustrates the elements of the signal array returned by the SS$_ ALIGN exception.

Figure 5–4 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 can impact performance for a user of your application. Once this exception is reported, your application has already experienced the performance impact.

5.4 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 allow 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 Alpha AXP architecture and between the calling standards for both architectures, the way in which many of these tasks are accomplished is not the same. Table 5–3 lists the run-time library condition handling support routines available on VAX systems and indicates which are supported on AXP systems.

Table 5–3 Run-Time Library Condition Handling Support Routines

Routine	Support on AXP 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 list provides specific guidelines for 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 will need to change your source code. These routines are not supported on AXP systems. Note, however, that certain types of arithmetic exceptions are always enabled on AXP systems. The following types of arithmetic exceptions are always enabled:
  - Floating-point invalid operation
  - Floating-point division by zero
  - Floating-point overflow

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

- If your application specifies a condition handling routine by calling the run-time library routine LIB$ESTABLISH, you may not have to change your source code. Most compilers on AXP systems, to preserve compatibility, accept calls to the LIB$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.

As an example, the program in Example 5–2, written in FORTRAN, 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.

To get this program to run on an AXP system, you must change the condition code from SS$_INTOVF to SS$_HPARITH. (You can determine the type of overflow by examining the exception summary argument in the signal array. For more information, see the compiler documentation.) As on 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 DEC Fortran accepts this routine as an intrinsic function.

Example 5–2 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)  1

        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

        INDEX = LIB$MATCH_COND (SIGARGS(2), SS$_INTOVF) 2
        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 items in the following list correspond to the numbered items in Example 5–2:

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

2 On an AXP system, you must change the condition code SS$_INTOVF to SS$_HPARITH. The handler routine calls the LIB$STOP routine to terminate execution of the program.

The following example illustrates how to compile, link, and run the program in Example 5–2:

```
$ FORTRAN/EXTEND_SOURCE/CHECK=OVERFLOW  HANDLER_EX.FOR
$ LINK  HANDLER_EX
$ RUN   HANDLER_EX
 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 Name   Module Name     Routine Name    Line Number  rel PC      abs PC
 INT_OVR_HAND INT_OVR_HANDLER HANDLER               1637 00000238    00020238
 DEC$FORRTL                                            0 000651E4    001991E4
----- above condition handler called with exception 00000504:
%SYSTEM-F-HPARITH, high performance arithmetic trap, Imask=00000001, Fmask=00000
000, summary=40, PC=000200E0, PS=0000001B
-SYSTEM-F-INTOVF, arithmetic trap, integer overflow at PC=000200E0, PS=0000001B
----- end of exception message
                                                       0 84FE9FFC    84FE9FFC
 INT_OVR_HAND INT_OVR_HANDLER INT_OVR_HANDLER         15 000000E0    000200E0
                                                       0 84EFD918    84EFD918
                                                       0 7FF23EE0    7FF23EE0
```

6 Ensuring Interoperability Between Native and Translated Images

This chapter describes how to create native AXP images that can make calls to and be called by translated VAX images.

6.1 Overview

DECmigrate for OpenVMS AXP Systems Translating Images describes how to use the VAX Environment Software Translator (VEST) to convert a VAX executable or shareable image into a functionally equivalent AXP image. (DECmigrate for OpenVMS AXP is an optional layered product that supports the migration of a VAX application to an AXP system. VEST is a component of the DECmigrate utility.)

Using VEST, you can translate all the components of an application, such as the main executable image and all the shareable images that it calls. However, you can also create an application that is a mix of translated and native components. For example, you may want to create a native version of a shareable image that is called by your application to take advantage of native performance. You may also choose to use a mixture of native and translated components to allow you to create a native version of your application in stages.

You can use translated VAX images as you would a native AXP image. However, to create native images that can interoperate with translated images requires some additional considerations, described in the following sections.

6.1.1 Compiling Native Images That Can Interoperate with Translated Images

To create a native image that can call or be called by a translated image, you must specify the /TIE qualifier when compiling the source files of the native AXP image. Any source module that contains a procedure that is made available to external callers must be compiled with the /TIE qualifier. When you specify the /TIE qualifier, the compilers create procedure signature blocks (PSBs) that are needed by the Translated Image Environment (TIE) at execution time in order to properly jacket calls between translated and native images. The TIE is part of the operating system.

You must also specify the /TIE qualifier when compiling a source module that contains a procedure that performs a callback (or calls out to a specified procedure) that may be in a translated image. In this case, the /TIE qualifier causes the compilers to generate a call to a special run-time library routine, OTS$CALL_PROC, that ensures that the outbound call to a translated procedure is handled properly.

In addition to the /TIE qualifier, you may need to specify other compiler qualifiers to ensure correct interoperation between a translated image and a native shareable image. For example, if the translated callers of a native shareable image use the VAX D_float format for double-precision floating-point operations (the default for VAX languages), you must specify the /FLOAT=D_FLOAT qualifier because the default format for double-precision data on AXP systems is *not* VAX D_floating. Consult compiler documentation to determine the exact qualifier syntax to specify VAX D_floating format. Note that, because the VAX D_floating data type is not supported by the AXP architecture, its use adversely affects performance.

Depending on application-specific semantics, you may also need to specify other compiler qualifiers to force byte granularity, data alignment, and AST atomicity.

6.1.2 Linking Native Images That Can Interoperate with Translated Images

To create a native AXP image that can call a translated VAX image, you must explicitly link the native object modules with the /NONATIVE_ONLY qualifier. (Note that /NATIVE_ONLY is the default used by the linker for this qualifier.) This qualifier causes the linker to include in the image the PSB information created by the compilers.

Because the /NONATIVE_ONLY qualifier affects only outgoing calls from native images to translated images, you do not need to specify it when creating a native AXP image that will be called by a translated VAX image. The linker's default behavior (/NATIVE_ONLY qualifier) can prevent native images from calling translated images but not from being called by translated images.

Note that the layout of the symbol vector in the native version of the shareable image must match the layout of the symbol vector in the translated shareable image it replaces. For more information about replacing translated shareable images with native shareable images, see Section 6.3.

6.2 Creating a Native Image That Can Call a Translated Image

To create a native AXP image that can make calls to a translated VAX shareable image, perform the following steps:

1. **Translate the VAX shareable image**. See *DECmigrate for OpenVMS AXP Systems Translating Images* for information about using VEST to translate VAX images.
2. **Create a native AXP version of the main program**. Compile the source modules using a compiler that produces native AXP code, specifying the /TIE qualifier on the command line.
3. **Link the native object module with the translated VAX shareable image**. Specify the translated image in a linker options file as you would any other shareable image.

To illustrate interoperability, consider the programs in Example 6–1 and Example 6–2. Example 6–1 calls the routine mysub that is defined in Example 6–2.

Example 6–1 Source Code for Main Program (MYMAIN.C)

```
#include <stdio.h>

int mysub();

main()
{
   int num1, num2, result;

   num1 = 5;
   num2 = 6;

   result = mysub( num1, num2 );
   printf("Result is: %d\n", result);
}
```

Example 6–2 Source Code for Shareable Image (MYMATH.C)

```
int myadd(value_1, value_2)
 int value_1;
 int value_2;
{
  int result;

  result = value_1 + value_2;
  return( result );
}

int mysub(value_1,value_2)
 int value_1;
 int value_2;
{
 int result;

 result = value_1 - value_2;
 return( result );
}

int mydiv( value_1, value_2 )
  int value_1;
  int value_2;
{
  int result;

  result = value_1 / value_2;
  return( result );
}

int mymul( value_1, value_2 )
  int value_1;
  int value_2;
{
  int result;

  result = value_1 * value_2;
  return( result );
}
```

To create VAX images from these examples, compile the source modules using a C compiler on a VAX system. To implement Example 6–2 as a shareable image, link the module, specifying the /SHAREABLE qualifier on the LINK command line and declaring the universal symbols in the shareable image by using the UNIVERSAL= option or by creating a transfer vector file. (See the *OpenVMS Linker Utility Manual* for information about how to create a shareable image.)

The following example illustrates a LINK command that creates the shareable image MYMATH.EXE:

```
$ LINK/SHAREABLE  MYMATH.OBJ,SYS$INPUT/OPT
GSMATCH=LEQUAL,1,1000
UNIVERSAL=myadd
UNIVERSAL=mysub
UNIVERSAL=mydiv
UNIVERSAL=mymul
Ctrl/Z
```

You can then link the main program with the shareable image to create the executable image MYMAIN.EXE, as in the following example:

```
$ LINK  MYMAIN.OBJ,SYS$INPUT/OPT
MYMATH.EXE/SHAREABLE
Ctrl/Z
```

Note that you may need to specify the /BPAGE qualifier on the LINK command line to force the linker to create image sections using a larger page size than the default page size on VAX systems (512 bytes). Otherwise, when VEST translates your VAX image, VEST may collect a number of these 512-byte image sections on a single AXP page. When VEST puts neighboring image sections with conflicting protection attributes on the same AXP page, it assigns the most permissive protection to all the image sections and issues a warning. (See the *OpenVMS Linker Utility Manual* for more information about using the /BPAGE qualifier.)

After creating the VAX images, translate them using VEST. Note that you must translate the shareable image first. (For more information about using the VEST command, see *DECmigrate for OpenVMS AXP Systems Translating Images*.) The following example creates the translated files named MYMATH_TV.EXE and MYMAIN_TV.EXE (VEST appends the characters "_TV" to the end of the image's file name):

```
$ VEST  MYMATH.EXE
$ VEST  MYMAIN.EXE
```

To replace the translated main executable image MYMAIN_TV.EXE with a native version, compile the source module in Example 6–1 using a compiler that generates AXP code, specifying the /TIE qualifier on the compile command line. Then link the native object module MYMAIN.OBJ to create a native AXP image, specifying the translated shareable image in the linker options file as you would any other shareable image, as in the following example:

```
$ LINK/NONATIVE_ONLY  MYMAIN.OBJ,SYS$INPUT/OPT
MYMATH_TV.EXE/SHAREABLE
Ctrl/Z
```

You can run the native main image as you would any other AXP image. Define the name of the translated shareable image, MYMATH_TV, as a logical name that points to the location of the translated shareable image (unless it is located in the directory pointed to by the SYS$SHARE logical name) and execute the RUN command, as in the following example:

```
$ DEFINE MYMATH_TV YOUR$DISK:[YOUR_DIR]MYMATH_TV.EXE
$ RUN MYMAIN
```

6.3 Creating a Native Image That Can Be Called by a Translated Image

To create a native AXP shareable image that can be called by a translated VAX image, perform the following steps:

1. **Translate the VAX shareable image**. Even though you are replacing the VAX version of the shareable image with a native version, you must still translate the shareable image to create a VEST interface information file (IIF). VEST needs the IIF associated with the shareable image when it translates an image that calls the shareable image. See *DECmigrate for OpenVMS AXP Systems Translating Images* for information about IIF files and about using VEST to translate VAX images. (Note that you may have to repeat this step to control the layout of the symbol vector in the translated shareable image. See Section 6.3.1 for more information.)
2. **Translate the VAX executable image that calls the shareable image**.
3. **Create a native AXP version of the shareable image**. Compile the source modules using a compiler that generates AXP code, specifying the /TIE qualifier on the command line.
4. **Link the object module to create a native AXP shareable image**. Use the SYMBOL_VECTOR= option to declare the universal symbols in the shareable image. For compatibility, declare the universal symbols in the SYMBOL_VECTOR= option in the same order as they were declared in the VAX shareable image.

Note

When creating a native AXP shareable image to replace a translated VAX shareable image, always leave the first entry of a symbol vector empty by specifying the SPARE keyword as the first entry in the SYMBOL_VECTOR= option. VEST reserves the first symbol vector entry in the translated VAX image for its own use.

The following example creates a native shareable image from the source module in Example 6–2:

```
$ LINK/SHAREABLE  MYMATH.OBJ,SYS$INPUT/OPT
GSMATCH=LEQUAL,1,1000  1
SYMBOL_VECTOR=(SPARE,-
              myadd=procedure,-  2
              mysub=procedure,-
              mydiv=procedure,-
              mymul=procedure)
Ctrl/Z
```

1 Specifies the major and minor identification numbers of the shareable image. The values of these identification numbers must match the values specified in the VAX shareable image. (For more information about using the GSMATCH= option, see the *OpenVMS Linker Utility Manual*.)

2 Specifies the universal symbols in the shareable image.

5. **Make sure the layout of the symbol vector in the native AXP image matches the symbol vector in the translated VAX image**. Section 6.3.1 describes how to determine the offsets of symbols in these symbol vectors and how to control the layout of these symbol vectors to ensure they match.

You can run the translated main image, MYMAIN_TV.EXE, with either the translated VAX shareable image, MYMATH_TV.EXE, or with the native AXP shareable image, MYMATH.EXE. By default, the translated executable image calls the translated shareable image. (The translated executable image contains a global image section descriptor [GISD] that points to the translated shareable image. The image activator activates the shareable images to which the image has been linked.)

To run the translated main image with the native shareable image, define the name of the shareable image MYMATH_TV as a logical name that points to the location of the native AXP shareable image, MYMATH.EXE, as in the following example:

```
$ DEFINE MYMATH_TV YOUR_DISK:[YOUR_DIR]MYMATH.EXE
$ RUN MYMAIN_TV
```

6.3.1 Controlling Symbol Vector Layout

To create a native AXP shareable image that can replace a translated VAX shareable image in an application, you must ensure that the universal symbols in the shareable images appear at the same offsets in the symbol vectors in both images. When a VAX shareable image is translated, VEST creates a symbol vector for the image that includes the universal symbols declared in the original VAX shareable image. (A translated image is actually an AXP image, created by VEST, and, like any other AXP shareable image, it lists universal symbols in a symbol vector.) To create a native shareable image that is compatible with a translated shareable image, you must make sure that the same symbols appear at the same offsets in the symbol vector in the native AXP shareable image and in the translated VAX shareable image it replaces.

To control how VEST lays out the symbol vector it creates in the translated VAX image, create a symbol information file (SIF) and include it in the translation operation. A SIF file is a text file with which you can specify the layout of entries in the symbol vector VEST creates for the translated image and to determine which symbols should appear in the global symbol table (GST) of the translated shareable image. If you do not specify the layout of the symbol vector, VEST may change the layout in subsequent retranslations of the shareable image. Note that VEST reserves the first symbol vector entry for its own use. For more information about SIF files, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

You control the layout of the symbol vector in a native shareable image by specifying the SYMBOL_VECTOR= option. The linker lays out the entries in a symbol vector in the order in which you specify the symbols in the SYMBOL_VECTOR= option statement. Make sure you list the symbols in the SYMBOL_VECTOR= option in the same order as they appear in the transfer vector used to create the VAX shareable image. For more information about using the SYMBOL_VECTOR= option, see the *OpenVMS Linker Utility Manual*.

To make sure the symbol vector in a translated shareable image matches the symbol vector in a native shareable image, perform the following steps:

1. **Translate the VAX shareable image, specifying the /SIF qualifier**. When you specify the /SIF qualifier, VEST generates a SIF file that lists the contents of the symbol vector. (For more information about creating and interpreting a SIF file, see *DECmigrate for OpenVMS AXP Systems Translating Images*.) The following example is the SIF file created by VEST for the shareable image MYMATH.EXE. Note that the entries start at the *second* position in the symbol vector (offset 10 hexadecimal):

```
! .SIF Generated by VEST (V1.0) on
! Image "MYMATH", "V1.0"
MYDIV                            00000018 +S +G 00000030    00 4e
MYSUB   1                        0000000c +S +G 00000020 2  00 4e
MYADD                            00000008 +S +G 00000010    00 4e
MYMUL                            00000010 +S +G 00000040    00 4e
```

1 The entry for the universal symbol MYSUB.

2 The offset of the entry for MYSUB in the translated image's symbol vector.

2. **Determine the symbol vector offsets in the native shareable image**. Use the ANALYZE/IMAGE utility to determine the offsets of the symbols in the symbol vector in the native shareable image. The following excerpt from an analysis of the shareable image MYMATH.EXE shows the offset of the symbol MYSUB:

```
   .
   .
   .
 4) Universal Symbol Specification (EGSD$C_SYMG)
  data type: DSC$K_DTYPE_Z (0)
  symbol flags:
   (0)  EGSY$V_WEAK      0
   (1)  EGSY$V_DEF       1
   (2)  EGSY$V_UNI       1
   (3)  EGSY$V_REL       1
   (4)  EGSY$V_COMM      0
   (5)  EGSY$V_VECEP     0
   (6)  EGSY$V_NORM      1
  psect: 0
  value: 16 (%X'00000020')
  symbol vector entry (procedure)
   %X'00000000 00010000'
   %X'00000000 00000050'
  symbol: "MYSUB"
   .
   .
   .
```

3. **Edit the offsets in the SIF file, if necessary**. Use a text editor to change the offsets listed in the SIF file if they do not match the offsets in the native shareable image. Remember that the first entry in the symbol vector is reserved for use by the VEST utility.

4. **Retranslate the VAX shareable image, including the SIF file in the translation operation**. In this translation operation, VEST creates the symbol vector in the translated image using the offsets specified in the SIF file. VEST looks for the SIF file in the current device and directory. (See *DECmigrate for OpenVMS AXP Systems Translating Images* for more information about using the VEST utility.)

6.3.2 Creating Stub Images

In some cases, it is not possible to completely replace a VAX shareable image with an AXP shareable image. For example, there may be functions in the VAX shareable image that are specific to the VAX architecture. In this situation, it may be necessary to build both a translated image and a native image that together provide the functionality of the original VAX shareable image. In other cases, there may be a need to define a relationship between a translated VAX shareable image and a new AXP shareable image. In both situations, the translated VAX image must be a **jacket image**.

When building a jacket image, create a stub version of the new AXP image on a VAX system. Then create a modified VAX shareable image that depends on it and translate it, specifying the /JACKET=*shrimg* qualifier, where *shrimg* is the name of the new AXP shareable image. Note that a throwaway translation of the stub image must be performed in advance so that there is an IIF file that describes it. For complete information about creating stub images, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

A
OpenVMS AXP Compilers

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

The following lists the compilers covered in this appendix (in alphabetical order) with pointers to the sections in which they are described.

- DEC Ada (Section A.1)
- DEC C (Section A.2)
- DEC COBOL (Section A.3)
- DEC Fortran (Section A.4)
- DEC Pascal (Section A.5)

A.1 Compatibility of DEC Ada Between AXP Systems and VAX Systems

DEC Ada includes nearly all the standard and extended Ada language features included in VAX Ada. These features are documented in the following manuals:

- *DEC Ada Language Reference Manual*
- *Developing Ada Programs on OpenVMS Systems*
- *DEC Ada Run-Time Reference Manual for OpenVMS Systems*

However, owing to differences in the platform hardware, some features are not supported or are implemented differently on VAX systems than on AXP systems. To aid in porting programs from one system to another, the following list highlights these differences.

Note

Not all of these features may be implemented on all systems for each release. See the DEC Ada release notes for more information.

A.1.1 Differences in Data Representation and Alignment

In general, DEC Ada supports the same data types on all platforms. However, keep in mind the following differences:

- H_float data

  Supported on VAX systems but not supported on AXP systems.

- IEEE floating formats

  Supported on AXP systems but not supported on VAX systems.

- Natural alignment

  On AXP systems, DEC Ada aligns record and array components on natural boundaries by default. On VAX systems, DEC Ada aligns record and array components on byte boundaries. Note that you can specify the alignment with the pragma COMPONENT_ALIGNMENT. The record representation clause maximum alignment is 2^9 on both VAX and AXP systems.

A.1.2 Tasking Differences

Task priorities and scheduling and task control block size are architecture specific. See the release notes for specifics.

A.1.3 Differences in Language Pragmas

Note the following differences in language pragmas:

- pragma COMPONENT_ALIGNMENT

  On AXP systems, COMPONENT_SIZE is the default choice. On VAX systems, STORAGE_UNIT is the default.

- pragma FLOAT_REPRESENTATION

  On AXP systems, this pragma supports two choices, IEEE_FLOAT and VAX_FLOAT. On VAX systems, this pragma supports the VAX_FLOAT choice.

- pragma LONG_FLOAT

  On AXP systems, the LONG_FLOAT pragma is supported when the value of the FLOAT_REPRESENTATION pragma is VAX_FLOAT.

- pragma SHARED

  There are type restrictions that are different between the systems. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information.

- pragma MAIN_STORAGE

  Not supported on AXP systems.

- pragma SHARE_GENERIC

  Not supported on AXP systems.

- pragma TIME_SLICE

  There are some implementation differences between the support of this pragma on VAX systems and on AXP systems. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information.

A.1.4 Differences in the SYSTEM Package

Note the following changes to the system package:

- SYSTEM.IEEE_SINGLE_FLOAT and SYSTEM.IEEE_DOUBLE_FLOAT

  Supported on AXP systems but not on VAX systems.

- SYSTEM.H_FLOAT

  Supported on VAX systems but not on AXP systems.

- SYSTEM.MAX_DIGITS

  The value is 33 on VAX systems and 15 on AXP systems.
- SYSTEM.NAME

  Specific enumerals are supported for each system on which DEC Ada is available.
- SYSTEM.SYSTEM_NAME

  The name OpenVMS_AXP is supported on AXP systems.
- SYSTEM.TICK

  The value is 10.0^{-3} (1 ms) on AXP systems. The value on VAX systems is 10.0^{-2} (10 ms).

In addition, the following types and subprograms, which are supported on VAX systems, are not supported on AXP systems:

SYSTEM.READ_REGISTER
SYSTEM.WRITE_REGISTER
SYSTEM.MFPR
SYSTEM.MTPR
SYSTEM.CLEAR_INTERLOCKED
SYSTEM.SET_INTERLOCKED
SYSTEM.ALIGNED_WORD
SYSTEM.ADD_INTERLOCKED
SYSTEM.INSQ_STATUS
SYSTEM.REMQ_STATUS
SYSTEM.INSQHI
SYSTEM.REMQHI
SYSTEM.INSQTI
SYSTEM.REMQTI

A.1.5 Differences Between Other Language Packages

Note the following differences in these other packages:

- package CALENDAR

  Implementation differences between systems; see the *DEC Ada Language Reference Manual* for more information.
- package MATH_LIB

  Implementation differences between systems; see individual package specifications.
- package SYSTEM_RUNTIME_TUNING

  This package is supported on VAX systems and, with some restrictions, on AXP systems. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* or the release notes for more information.

A.1.6 Changes to Predefined Instantiations

The following two predefined instantiations, which are supported on VAX systems, are not supported on AXP systems:

- LONG_LONG_FLOAT_TEXT_IO
- LONG_LONG_FLOAT_MATH_LIB

A.2 Compatibility of DEC C for OpenVMS AXP Systems with VAX C

To support the Alpha AXP architecture, a compiler is being added to the set of C compilers known collectively as DEC C. The set of compilers that comprise DEC C define a core, ANSI-conforming C language that can be used on all strategic Digital platforms, including the Alpha AXP architecture.

A.2.1 Language Modes

DEC C for OpenVMS AXP systems conforms to the ANSI C standard, with optional support for VAX C and Common C (pcc) extensions. You invoke these optional extensions, called **modes**, using the /STANDARD qualifier. Table A–1 describes these modes and the command-qualifier syntax needed to invoke them.

Table A–1 Modes of Operation of the DEC C for OpenVMS AXP Systems

Mode	Command Qualifier	Description
Default	/STANDARD=RELAXED_ANSI89	Follows ANSI C standard but also allows additional Digital keywords and predefined macros that do not begin with an underscore
ANSI C	/STANDARD=ANSI89	Accepts only strictly conforming ANSI C language
VAX C	/STANDARD=VAXC	Allows use of VAX C extensions to the ANSI C standard, even where the extensions may be incompatible with the ANSI C standard
Common C (pcc)	/STANDARD=COMMON	Allows use of Common C extensions to the ANSI C standard, even where the extensions may be incompatible with the ANSI C standard
Combination of VAX C and Common C	/STANDARD=(VAXC,COMMON)	Allows use of both VAX C and Common C extensions to the ANSI C standard, even where the extensions may be incompatible with the ANSI standard

A.2.2 DEC C for OpenVMS AXP Systems Data-Type Mappings

The DEC C for OpenVMS AXP systems compiler supports most of the same data-type mappings as its VAX counterpart. Table A–2 lists the sizes of the C arithmetic data types on the Alpha AXP architecture.

Table A–2 Arithmetic Data-Type Sizes in DEC C for OpenVMS AXP Compiler

C Data Type	VAX C Mapping	DEC C Mapping
pointer	32	32 or 64[1]
long	32	32
int	32	32
short	16	16
char	8	8

[1]When implemented, you will be able to select the size by using a pragma in your source file or by using a command line qualifier.

(continued on next page)

Table A–2 (Cont.) Arithmetic Data-Type Sizes in DEC C for OpenVMS AXP Compiler

C Data Type	VAX C Mapping	DEC C Mapping
float	32	32[2]
double	64[2]	64[2]
long double	64[2]	64[2]
__int16	NA	16
__int32	NA	32
__int64	NA	64

[2]You select how this maps to an AXP D, F, G, S, or T floating point by using a command line qualifier. See Section A.2.2.1.

To aid portability, the DEC C for OpenVMS AXP systems compiler provides a header file that defines macros for each data type. These macros map a generic data-type name, such as int64, to the machine-specific data type, such as *–64*. For example, if you must have a data type that is 64 bits long, use the int64 macro.

A.2.2.1 Specifying Floating-Point Mapping

The mapping between the C floating-point data types and the Alpha AXP floating-point data types is controlled by command line qualifiers. The Alpha AXP architecture supports the following floating-point types:

- F_floating (same as on OpenVMS VAX systems)
- D_floating (53-bit precision)
- G_floating (same as on OpenVMS VAX systems)
- S_floating (IEEE single precision)
- T_floating (IEEE double precision)

By using a command line qualifier, you control which of the Alpha AXP floating-point data types the standard C data types float and double map to. For example, if you specify the /FLOAT=G_FLOAT qualifier, DEC C maps the float data type to the Alpha AXP F_float data type and maps the double data type to the Alpha AXP G_float data type. Table A–3 describes the complete list of floating-point options. Note that you can specify only one floating-point qualifier in a command line.

Table A–3 DEC C Floating-Point Mappings

Compiler Option	Float	Double
/FLOAT=F_GLOAT	F_float format	G_float format
/FLOAT=D_FLOAT	F_float format	D-53 floating point
/FLOAT=IEEE_FLOAT	S_float format	T_float format

A.2.3 Features Specific to AXP Systems

DEC C includes features, summarized in Table A–4, that are specific to AXP systems. The following sections describe these features.

Table A–4 DEC C Compiler Features Specific to AXP Systems

Feature	Description
Access to some Alpha AXP instructions	Available as built-ins
Access to some VAX instruction equivalents	Available through Alpha AXP PALcode
Atomicity built-ins	Ensures the atomicity of AND, OR, and ADD operations

A.2.3.1 Accessing Alpha AXP Instructions

DEC C supports certain Alpha AXP instructions to provide functions that cannot be expressed in the C language, especially for system-level programming. Currently, DEC C plans to support the following Alpha AXP instructions:

- TRAPB—Drain the instruction pipeline
- MB—Memory barrier

A.2.3.2 Accessing Alpha AXP Privileged Architecture Library (PALcode) Instructions

The Alpha AXP architecture implements certain VAX instructions as privileged architecture library (PALcode) instructions. DEC C allows access to the following PALcode instructions:

- INSQUEx—Insert entry into longword or quadword queue
- INSQxI—Insert entry in queue, interlocked
- REMQUEx—Remove entry from longword or quadword queue
- REMQxI—Remove from queue, interlocked

Note, however, that the following VAX instructions that are supported as built-ins in VAX C are not supported as built-ins by DEC C:

ADAWI	BBCCI	BBSSI	FFC
FFS	LDPCTX	LOCC	MFPR
MTPR	MOVC3	MOVPSL	PROBER
PROBEW	READ_GPR	SCANC	SIMPLE_READ
SKPC	SPANC	SCSVPCTX	WRITE_GPR

A.2.3.3 Ensuring the Atomicity of Combined Operations

In the VAX architecture, certain combined operations, such as incrementing a variable, are guaranteed to be atomic (that is, they complete without interruption). To provide an equivalent function on AXP systems, DEC C provides built-ins that perform the operations with the guarantee of atomicity. Table A–5 lists these atomic built-ins.

Table A–5 Atomicity Built-Ins

Atomicity Built-In	Description

__ADD_ATOMIC_LONG(ptr, expr, retry_count)
__ADD_ATOMIC_QUAD(ptr, expr, retry_count)
	Add the expression **expr** to the data segment pointed to by **ptr**. The optional **retry_count** parameter specifies the number of times the operation should be attempted (the default is forever).

__AND_ATOMIC_LONG(ptr, expr, retry_count)
__AND_ATOMIC_QUAD(ptr, expr, retry_count)
	Fetch the data segment pointed to by **ptr**, perform a logical AND operation with the expression **expr**, and store the resulting value. The **retry_count** parameter specifies the number of times the operation should be attempted (the default is forever).

__OR_ATOMIC_LONG(ptr, expr, retry_count)
__OR_ATOMIC_QUAD(ptr, expr, retry_count)
	Fetch the data segment pointed to by **ptr**, perform a logical OR operation with the expression **expr**, and store the resulting value. The **retry_count** parameter specifies the number of times the operation should be attempted (the default is forever).

These built-ins guarantee only that the operation completes without interruption. If you perform an atomic operation on a variable that might be subject to concurrent write access (for example, from an AST and mainline code or from two concurrent processes), you must still protect it with the **volatile** attribute.

In addition, DEC C for OpenVMS AXP systems supports the following equivalents to the VAX interlocked instructions:

- TESTBITSSI
- TESTBITCCI

These built-ins use the **retry_count** parameter, as do the atomicity built-ins, to avoid getting stuck in an endless loop.

A.2.4 Differences Between the VAX C and DEC C for OpenVMS AXP Systems Compilers

The following features, present in VAX C, have different default behavior in DEC C for OpenVMS AXP systems. Note, however, that for some of these features, you can retain the VAX C behavior by using command line qualifiers and pragma instructions.

A.2.4.1 Controlling Data Alignment

Because accesses to data that is not aligned on natural boundaries cause severe performance degradation on AXP systems, DEC C for OpenVMS AXP systems aligns data on natural boundaries by default. To override this feature and retain VAX (packed) alignment, specify the nomember_alignment pragma in your source file or use the /NOMEMBER_ALIGNMENT command line qualifier.

A.2.4.2 Accessing Argument Lists

Taking the address of an argument, such as &argv1, causes DEC C for OpenVMS AXP systems to generate prologue code for the function that moves all the arguments onto the stack (called **homing** arguments), causing a performance degradation. Also, argument list "walking" can be accomplished only by using the functions in the VARARGS.H or STDARGS.H include files.

A.2.4.3 Synchronizing Exceptions

Because the Alpha AXP architecture does not provide for immediate reporting of arithmetic exceptions, do not expect an assignment to a static variable (even with the **volatile** attribute) to occur before a subsequent exception is signaled.

A.2.5 VAX C Features Not Supported by /STANDARD=VAXC Mode

While most programming practices supported by VAX C are supported by DEC C for OpenVMS AXP systems in /STANDARD=VAXC mode, certain programming practices that conflict with the ANSI standard are not supported. The following list highlights some of these differences; see the DEC C compiler documentation for more information.

- The inclusion of text after an #endif statement, as in the following example:

  ```
  #ifdef a
     .
     .
     .
  #endif a
  ```

  Delete the text or surround it with comment delimiters, as in the following:

  ```
  #endif /* a */
  ```

- Modification of string constants, while always a questionable programming practice, was accepted by VAX C. DEC C for OpenVMS AXP systems places all string constants in a read-only program section so that they cannot be modified.

- Structure-initialization values must be enclosed within braces ({}):

  ```
  array[SIZE] = NULL;   /* accepted by VAX C */
  array[SIZE] = {NULL}; /* required by DEC C */
  ```

- Redefinitions of symbols are now flagged with a warning-level diagnostic message:

  ```
  #define x  a
  #define x  b   /* generates a warning message in DEC C */
  ```

- Use of text libraries is no longer recommended. While supported by VAX C, text libraries are not portable.

  ```
  #include  stdio
  ```

  Instead, use the following syntax:

  ```
  #include <stdio.h>
  ```

- You must have one, and only one, declaration of an external variable. This is the definition of this variable. Other modules can use it by declaring it with the `extern` semantics.

A.3 Compatibility of DEC COBOL with VAX COBOL

The DEC COBOL Version 1.0 compiler, running on an OpenVMS AXP system, is based on and is highly compatible with the VAX COBOL Version 4.4 compiler running on an OpenVMS VAX system. The DEC COBOL compiler supports many, but not all, VAX COBOL features. The following list summarizes some of the major differences betweeen the DEC COBOL and VAX COBOL compilers:

- A new alignment qualifier that selects Alpha AXP data alignment to optimize performance or VAX COBOL data alignment to ensure compatibility with VAX COBOL record alignment
- A new qualifier that provides both IEEE and VAX floating-point data types for single- and double-precision data items
- A new qualifier to generate code that allows native images to call translated images and translated images to call native images
- A new qualifier to recognize additional COBOL reserved words defined by the X/Open Portability Guide
- A new screen manager for ACCEPT/DISPLAY extensions
- Support for only the most important features of the VAX COBOL /STANDARD=V3 qualifier option
- No support for the VAX DBMS (Database Management System) Data Manipulation Language (DML)
- No support for intrinsic functions, which are included in VAX COBOL Version 5.0 and higher
- No support for multibyte characters and other Japanese-language features, which are included in Version 5.0 and higher of VAX COBOL (Japanese version)
- Support for file status values that are compatible with VAX COBOL Version 5.1, which differ from those of VAX COBOL Version 5.0 and previous versions

The information provided in this section is intended to help you write COBOL applications that are compatible with both VAX COBOL and DEC COBOL as well as to help you convert your existing COBOL applications from VAX COBOL to DEC COBOL.

This section describes similarities and differences between VAX COBOL Version 4.4 and DEC COBOL Version 1.0. Differences between DEC COBOL and later versions of VAX COBOL are noted when warranted.

For the latest information about product features and future release enhancements of the DEC COBOL compiler, refer to the most recent version of the DEC COBOL release notes. For information about VAX COBOL features, refer to the VAX COBOL release notes and other documentation. You can obtain an online version of the release notes for your installed COBOL compiler by entering the HELP COBOL RELEASE_NOTES command at the system prompt.

For reference information about DEC COBOL language features, see the *DEC COBOL Reference Manual*. For reference information about VAX COBOL language features, see the *VAX COBOL Reference Manual*. For information about DEC COBOL command line qualifiers, invoke the online help system for COBOL at the operating system prompt. For information about VAX COBOL command line qualifiers, see the *VAX COBOL User Manual*.

A.3.1 Command Line Qualifiers

Tables A–6, A–7, and A–8 compare and contrast the DEC COBOL and VAX COBOL command line qualifiers.

A.3.1.1 Qualifiers Shared by DEC COBOL and VAX COBOL

Table A–6 lists the command line qualifiers shared by DEC COBOL and VAX COBOL. For more information about the command line qualifiers available in DEC COBOL, refer to Table A–7 or invoke the DEC COBOL online help system. For more information about the VAX COBOL command line qualifiers, refer to Table A–8 and the *VAX COBOL User Manual*.

Table A–6 Qualifiers and Options Shared by DEC COBOL and VAX COBOL

Qualifier	Comments
/ANALYSIS_DATA	Equivalent.
/ANSI_FORMAT	Equivalent.
/AUDIT	Equivalent.
/CHECK	A new option (/CHECK=[NO]DECIMAL) is available for DEC COBOL. (See Table A–7 and Section A.3.2.2.)
/CONDITIONALS	Equivalent.
/COPY_LIST	Equivalent.
/CROSS_REFERENCE	Equivalent.
/DEBUG	Equivalent.
/DEPENDENCY_DATA	Equivalent.
/DIAGNOSTICS	Equivalent.
/FIPS	Minor differences in functionality exist. (Invoke the DEC COBOL online help system for information about the behavior of the /FIPS=74 qualifier option.)
/FLAGGER	Equivalent.
/LIST	Equivalent.
/MACHINE_CODE	Equivalent.
/MAP	Equivalent.
/OBJECT	Equivalent.
/SEQUENCE_CHECK	Equivalent.
/STANDARD	Some VAX COBOL options are available in DEC COBOL. (See Section A.3.2.7 for information about the behavior of the /STANDARD=V3 qualifier option.)
/TRUNCATE	Equivalent.
/WARNINGS	Minor differences in functionality exist. (See Section A.3.2.7.2 and invoke the DEC COBOL online help system for information about the behavior of the /WARNINGS qualifier.)

A.3.1.2 DEC COBOL Qualifiers Not Available in VAX COBOL

Table A–7 lists the command line qualifiers and options that are specific to DEC COBOL. These qualifiers and options are not available in VAX COBOL. For more information about the command line qualifiers available in DEC COBOL, invoke the DEC COBOL online help system.

Table A–7 DEC COBOL Qualifiers Not Available in VAX COBOL

Qualifier	Comments
/ALIGNMENT=([NO]BINARY, [NO]DECIMAL)	Specifies the data alignment for numeric data items. (See Section A.3.2.1.)
/CHECK=[NO]DECIMAL	Validates numeric digits when using display numeric items in a numeric context. (See Section A.3.2.2.)
/CONVERT=LEADING_BLANKS	Changes leading blanks to zeros in numeric display items. (See Section A.3.2.3.)
/FLOAT=[D_FLOAT],[IEEE_ FLOAT]	Specifies the floating-point data format to be used in memory for single- and double-precision data items. (See Section A.3.2.4.)
/OPTIMIZE	Controls whether the compiler optimizes the compiled program to generate more efficient code. (See Section A.3.2.5.)
/RESERVED_WORDS=[NO]XOPEN	Controls whether or not the compiler recognizes X/Open COBOL words as reserved words. (See Section A.3.2.6.)
/TIE	Generates code that allows native images to call translated images and translated images to call native images. (See Section A.3.2.8.)

A.3.1.3 VAX COBOL Qualifiers Not Available in DEC COBOL

Table A–8 lists the command line qualifiers and options that are specific to VAX COBOL. These qualifiers and options are not available in DEC COBOL. For detailed information about the VAX COBOL command line qualifiers, refer to the *VAX COBOL User Manual*.

Table A–8 VAX COBOL Qualifiers Not Available in DEC COBOL

Qualifier	Comments
/DESIGN	Controls whether the compiler processes the input file as a detailed design.
/INSTRUCTION_SET[=option]	Improves run-time performance on single-chip VAX processors, using different portions of the VAX instruction set.
/STANDARD=[NO]OPENVMS_AXP	Produces informational messages about language features that are not supported by the DEC COBOL compiler. (See Section A.3.2.9 and the VAX COBOL Version 5.1 release notes.)
/STANDARD=[NO]PDP11	Produces informational messages about language features that are not supported by the COBOL–81 compiler.
/WARNINGS=[NO]STANDARD	Produces informational messages about language features that are Digital extensions. The DEC COBOL equivalent is /STANDARD=[NO]SYNTAX. (See Section A.3.2.7.2.)

A.3.2 Behavior Differences

This section describes differences in behavior between VAX COBOL Version 4.4 and DEC COBOL Version 1.0, including new DEC COBOL command line qualifiers and options, as well as behavior that is specific to DEC COBOL Version 1.0.

A.3.2.1 Specifying Alignment for Numeric Data Items with the DEC COBOL /ALIGNMENT Qualifier and Alignment Directives

You can use the /ALIGNMENT qualifier and alignment directives to specify the alignment of binary and decimal data items within record structures. Refer to the *DEC COBOL Reference Manual* for specific information about alignment.

Proper data alignment is needed to optimize your COBOL applications on Alpha AXP systems. Manipulating binary data is significantly faster if the data lies within natural boundaries. Just as important, manipulating decimal data is significantly faster if you align the data along the preferred boundaries for the system you are using.

The primary goal of alignment specification is optimum performance. In addition, the /ALIGNMENT qualifier and alignment directives meet the following goals:

- Ease of use and conversion—You need to make only a minimal number of changes to your existing source files. In some cases, all you need to do is add the /ALIGNMENT qualifier when you invoke the DEC COBOL compiler.
- VAX COBOL source compatibility—You can compile the same source files with VAX COBOL and DEC COBOL. DEC COBOL directives are structured comments that the VAX COBOL compiler ignores.
- Flexibility—You can specify VAX byte alignment or natural alignment on a record-by-record basis. For example, you can specify byte alignment for files that are shared by both compilers and natural alignment for DEC COBOL files and records.

The /ALIGNMENT qualifier, alignment directives, and the SYNCHRONIZED clause affect the alignment of items within a group (group elements) as shown in the following figure:

```
                 /ALIGNMENT qualifier
                           |
 Alignment directive       |                  SYNCHRONIZED clause
           |               |                          |
           |_______________|________                  |
                                    |                 |
                                    |                 |
                 Align binary and decimal    Align binary
                 group elements              group elements
```

As with the VAX COBOL compiler, you can use the SYNCHRONIZED clause to align binary components of records on natural boundaries. Thus, for the DEC COBOL compiler operating on binary data, the SYNCHRONIZED clause, the /ALIGNMENT qualifier, and alignment directives can exhibit equivalent behavior.

A.3.2.1.1 Using the /ALIGNMENT Qualifier

The /ALIGNMENT qualifier allows you to specify natural alignment for binary data and preferred alignment for numeric decimal data in your program.

Binary and decimal alignment are separate options (a useful feature for programs that alias decimal and string data, but that can still benefit from the alignment of binary data). For example, when you specify /ALIGNMENT=(BINARY,NODECIMAL) (or /ALIGNMENT), the DEC COBOL compiler aligns binary data along natural boundaries and decimal data along byte boundaries. Use /ALIGNMENT to ensure that your data is aligned for optimum performance on OpenVMS AXP systems.

Use /NOALIGNMENT, the default, to specify byte data alignment (including programs that align binary data items with the SYNCHRONIZED clause). Also use /NOALIGNMENT for portability and compatibility with data files produced on an OpenVMS VAX system.

A.3.2.1.2 Using Alignment Directives

The alignment properties specified by the /ALIGNMENT qualifier remain in effect throughout a given compilation, except as modified by alignment directives. **Directives** are structured comments that the DEC COBOL compiler interprets. (DEC COBOL directives are ignored by the VAX COBOL compiler.) All directives begin with "*DC", where the "*" signals the beginning of the structured comment.

You can use the following alignment directives anywhere within your COBOL source program to change the current set of alignment porperties:

- *DC SET ALIGNMENT[=(option,...)] (where *option* is [NO]BINARY or [NO]DECIMAL)—Specifies a new alignment. Specifying *DC SET ALIGNMENT is equivalent to specifying *DC SET ALIGNMENT=(BINARY,NODECIMAL).
- *DC END-SET ALIGNMENT—Restores the alignment to the previous setting. (Use of this alignment directive is optional.)
- *DC SET NOALIGNMENT—Specifies byte alignment.

You can nest alignment directives within your program to turn alignment on or off for specific numeric data items. Although the *DC END-SET ALIGNMENT directive is optional, you must use it to indicate the end of each nested alignment directive.

A.3.2.2 Validating Numeric Data with the DEC COBOL /CHECK=NODECIMAL Qualifier Option

The /CHECK=[NO]DECIMAL qualifier option validates numeric characters when you use display numeric items in a numeric context. Use /CHECK=DECIMAL when you want the system to generate an error for any invalid, or nonnumeric, characters.

This feature is primarily intended to help validate data produced by other systems that might use a different internal representation for numeric data. A secondary consideration is that this qualifier can also be used to detect logic errors in programs that result in text data being moved to numeric data items. The disadvantage of this feature is that extra instructions are needed to perform the checks, resulting in slightly larger images and slightly longer execution times.

Use /CHECK=NODECIMAL, the default, when you do not want the system to check for numeric characters in numeric display items.

A.3.2.3 Converting Leading Blanks to Zeros with the DEC COBOL /CONVERT=LEADING_BLANKS Qualifier Option

The /CONVERT=LEADING_BLANKS qualifier and option generates code to check for and change leading blanks to zeros in numeric display items.

This feature is primarily intended to help users convert existing COBOL programs to run on an OpenVMS AXP system by changing leading blanks in the data to zeros at run time. The disadvantage of this feature is that extra instructions are needed to perform the data conversions. This results in slightly larger images and slightly longer execution times.

Use /NOCONVERT=LEADING_BLANKS, the default, when you do not want the compiler to change leading blanks to zeros in numeric display items.

A.3.2.4 Specifying a Floating-Point Data Format with the DEC COBOL /FLOAT Qualifier

The /FLOAT=[option] qualifier specifies the floating-point data format to be used in memory for single- and double-precision data items. Specify either /FLOAT=D_FLOAT or /FLOAT=IEEE_FLOAT within a single program.

Because the Alpha AXP architecture is IEEE-compliant, you can run existing COBOL programs containing IEEE floating-point data formats on DEC COBOL.

Use the /FLOAT=D_FLOAT qualifier option, the default, at compile time to specify the VAX F_floating memory format for single-precision (COMP-1) data and the VAX D_floating memory format for double-precision (COMP-2) data.

The IEEE standard for binary floating-point arithmetic, ANSI/IEEE 754-1985, defines four floating-point formats in two groups, basic and extended, each group having two widths, single and double. The Alpha AXP architecture supports the basic single and double formats.

Use the /FLOAT=IEEE_FLOAT qualifier option at compile time to specify the IEEE S_floating memory format for single-precision (COMP-1) data and the IEEE T_floating memory format for double-precision (COMP-2) data.

Refer to the *Alpha Architecture Handbook* for more information about using floating-point data types with the Alpha AXP architecture.

A.3.2.5 Optimizing Your Code with the DEC COBOL /OPTIMIZE Qualifier

The /OPTIMIZE qualifier controls whether the compiler optimizes the compiled program to generate more efficient code.

Use /OPTIMIZE, the default, when you want your program to run faster. Note that using this qualifier may cause the compiler to produce larger object modules and result in longer compile times.

Use /NOOPTIMIZE for a debugging session to ensure that the machine code occurs in the same order as the program lines in your source program.

A.3.2.6 Checking for Special Reserved Words with the DEC COBOL /RESERVED_WORDS Qualifier

The /RESERVED_WORDS qualifier controls whether or not the compiler recognizes certain COBOL words as reserved words.

Use /RESERVED_WORDS=NOXOPEN if your program uses one or more of the COBOL words defined by the X/Open Portability Guide as an identifier.

Use /RESERVED_WORDS=XOPEN, the default, if none of the following X/Open COBOL words appears in your program:

AUTO
BACKGROUND-COLOR
BELL
BLINK
EOL
EOS
ERASE
FOREGROUND-COLOR
FULL
HIGHLIGHT
LOWLIGHT
REQUIRED
REVERSE-VIDEO
SCREEN
SECURE
UNDERLINE

A.3.2.7 Calling Out Language Feature Extensions to the COBOL ANSI Standard with the DEC COBOL /STANDARD Qualifier

The /STANDARD qualifier controls whether the compiler prints informational messages associated with specific language features. To receive these messages, specify /STANDARD or /STANDARD=85 (and /WARNINGS=ALL or /WARNINGS=INFORMATIONAL) or /STANDARD=SYNTAX.

Use /STANDARD=85, the default, to instruct the DEC COBOL compiler to compile and generate code according to the ANSI 1985 COBOL standard.

Use /STANDARD=SYNTAX to instruct the DEC COBOL compiler to produce informational messages about language features that are Digital extensions to the ANSI 1985 COBOL Standard. The default, NOSYNTAX, suppresses these messages.

Use /STANDARD=V3 to instruct the DEC COBOL compiler to compile and generate code in the manner of VAX COBOL Version 3.4 in specific instances. Section A.3.2.7.1 describes the /STANDARD=V3 qualifier option in more detail.

A.3.2.7.1 /STANDARD=V3 Qualifier Option DEC COBOL Version 1.0, as with VAX COBOL Version 4.0 and higher versions, is based on the ANSI 1985 COBOL standard. As such, DEC COBOL provides full support for the /STANDARD=85 qualifier option. DEC COBOL also provides support for some features of the /STANDARD=V3 qualifier option that were available with VAX COBOL Version 4.0 and higher.

VAX COBOL versions prior to Version 4.0 were based on the ANSI 1974 COBOL standard. While most of the enhancements made to VAX COBOL Version 4.0 and higher versions are compatible with earlier versions of the VAX COBOL compiler, some differences exist, which causes results to vary in some instances.

To minimize conflicts with existing VAX COBOL programs, VAX COBOL allows you to compile programs according to the rules for either VAX COBOL Version 4.0 and later versions or according to the rules for VAX COBOL Version 3.4. Specifying /STANDARD=V3 instructs the VAX COBOL compiler to compile and generate code in the manner of VAX COBOL Version 3.4 in specific instances, as described in the *VAX COBOL User Manual*.

When compared with the features that are available with VAX COBOL Version 4.0 and higher, DEC COBOL provides limited support for the /STANDARD=V3 qualifier option. When you specify /STANDARD=V3, the DEC COBOL behavior is identical to the VAX COBOL Version 4.0 and higher behavior in the following four specific instances:

- EXIT PROGRAM statement in a main program
- I-O file status codes
- No valid next record condition
- Opening nonoptional files in I-O and EXTEND mode

The following four subsections describe this DEC COBOL behavior in more detail.

EXIT PROGRAM Statement

If you specify /STANDARD=V3, an EXIT PROGRAM statement is treated as a return in both main programs and subprograms.

Specifying /STANDARD=85 bypasses an EXIT PROGRAM statement in the body of a main program and executes the statements following the EXIT PROGRAM statement. If the program is a subprogram, the EXIT PROGRAM statement acts as a return to the program that called the subprogram.

I-O File Status Codes

If you specify /STANDARD=V3, you receive the file status codes listed in the left-hand column, labeled V3, and your program acts accordingly.

If you specify /STANDARD=85, you receive the file status codes listed in the right-hand column, labeled 85, and your program acts accordingly.

Table A–9 explains the I-O file status codes for VAX COBOL Version 3.4 and DEC COBOL.

Table A–9 I-O File Status Codes for the /STANDARD Qualifier

I-O Error Condition	Status Code	
	V3	85
READ successful—record shorter than fixed file attribute.	00	04
CLOSE reel/unit attempted on nonreel/unit device.	00	07
READ fails—relative key digits exceed relative key.	00	14
WRITE fails—relative key digits exceed relative key.	00	24
OPEN I-O on file that is not mass storage.	00	37
WRITE fails—attempt to write a record of a different size than in the file description.	00	44
READ fails—no next logical record (EOF detected).	13	10
READ fails—no next logical record (EOF on OPTIONAL file).	15	10
READ fails—no valid next record (already at EOF).	16	10
READ NEXT or sequential READ—no valid next record pointer.	16[1]	46[1]
READ or START fails—optional input file not present.	25	23

[1]See the subsection No Valid Next Record Condition.

(continued on next page)

Table A–9 (Cont.) I-O File Status Codes for the /STANDARD Qualifier

I-O Error Condition	Status Code	
	V3	85
READ successful—record longer than fixed file attribute.	30	04
OPEN on relative or indexed file that is not mass storage.	30	37
REWRITE fails—attempt to rewrite record of different size.	30	44
CLOSE fails—file not currently open.	93	42
DELETE or REWRITE fails—previous I-O not successful READ.	93	43
OPEN fails—file previously closed with LOCK.	94	38
OPEN fails—file created with different organization.	94	39
OPEN fails—file created with different prime record key.	94	39
OPEN fails—file created with different alternate record keys.	94	39
OPEN fails—file currently open.	94	41
READ or START fails—file not opened INPUT or I-O.	94	47
WRITE fails—file not opened OUTPUT, EXTEND, or I-O.	94	48
DELETE or REWRITE fails—file not opened I-O.	94	49
OPEN INPUT on a nonoptional file—file not found.	97	35

No Valid Next Record Condition

This subsection describes what happens when you compile your program using either /STANDARD=V3 or /STANDARD=85 and when all the following conditions exist:

- The no valid next record (NVNR) condition exists.
- Your program attempts a sequential READ statement.
- Your program includes an AT END branch associated with the READ statement.

When you use /STANDARD=V3 to compile your program, the following occurs:

- The file status code variable, if any, for the file is set to 16.
- The statements associated with the AT END statement are executed.
- The program continues to execute normally.

If you use /STANDARD=85 to compile your program, the following occurs:

- The file status code variable, if any, for the file is set to 46.
- The statements associated with the AT END statement are not executed.
- The program terminates execution abnormally (unless you have provided for this situation with a USE AFTER STANDARD EXCEPTION procedure).

OPEN I-O and EXTEND Modes

If you specify /STANDARD=V3, nonoptional files opened in
I-O or EXTEND mode are created, if the files are unavailable.

If you specify /STANDARD=85, nonoptional files opened in I-O or EXTEND mode are not created if the files are unavailable. Instead, a run-time error is issued.

A.3.2.7.2 /STANDARD and /WARNINGS Qualifiers VAX COBOL provides two qualifiers that specify the same behavior: /STANDARD=[NO]SYNTAX and /WARNINGS=[NO]STANDARD.

DEC COBOL does not support the [NO]STANDARD option of the /WARNINGS qualifier. Therefore, specifying /WARNINGS=ALL with the DEC COBOL compiler will not produce the informational messages that point out Digital extensions. To receive messages such as the following one, you must specify /STANDARD=SYNTAX.

```
%COBOL-I-EXTENSION
```

Note

For VAX COBOL and DEC COBOL, the FIPS messages about Digital extensions that the compiler produces when you specify /FLAGGER[(=option, . . .)] continue to be controlled by the /WARNINGS=INFORMATION qualifier option.

A.3.2.8 Calling Native and Translated Images with the DEC COBOL /TIE Qualifier

The /TIE (Translated Image Environment) qualifier generates code that allows native OpenVMS AXP images to call translated images and translated images to call native OpenVMS AXP images. This qualifier is supported on OpenVMS AXP systems only.

Specifying /TIE enables you to use compiled code with shared translated images, either because the code might call into a translated image or because it might be called from a translated image. If you specify /TIE, you should link the object module using the LINK command qualifier /NONATIVE_ONLY. (See the *OpenVMS Linker Utility Manual* for information about the /NONATIVE_ONLY qualifier.)

Specifying /NOTIE, the default, indicates that your compiled code will not be associated with a translated image.

For information about interoperability, see Chapter 6. For information about translated images, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

A.3.2.9 VAX COBOL to DEC COBOL Program Conversion

VAX COBOL Version 5.1 provides a new flagging system, via the /STANDARD=OPENVMS_AXP qualifier option, to identify language features in your existing VAX COBOL programs that are not available in DEC COBOL on OpenVMS AXP.

When you specify /STANDARD=OPENVMS_AXP (and /WARNINGS=ALL or /WARNINGS=INFORMATIONAL), the VAX COBOL compiler generates informational messages to flag language constructs that are not available in DEC COBOL. You can use this information to modify your program before running it on DEC COBOL.

Use /STANDARD=NOOPENVMS_AXP, the default, to suppress these informational messages.

A.3.2.10 Program Structure

In some cases, the DEC COBOL compiler generates more complete messages about unreachable code or other logic errors than does the VAX COBOL compiler.

The following example illustrates a sample program and the messages issued by the DEC COBOL compiler.

Source file:

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. T1.
       ENVIRONMENT DIVISION.
       PROCEDURE DIVISION.
       P0.
           GO TO P1.
       P3.
           GO TO P2.
       P2.
           DISPLAY "This is unreachable code".
       P1.
           STOP RUN.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. T2.
       ENVIRONMENT DIVISION.
       PROCEDURE DIVISION.
       P0.
           DISPLAY "This is unreachable code".
           EXIT PROGRAM.
       END PROGRAM T2.
       END PROGRAM T1.
```

On VAX systems:

```
$ COBOL /ANSI/WARNINGS=ALL T1.COB
```

On AXP systems:

```
$ COBOL/ANSI/OPT/WARNINGS=ALL T1.COB
       PROGRAM-ID. T2.
..................^
%COBOL-I-UNCALLED, routine T2 can never be called
at line number 14 in file DISK$YOURDISK:[TESTDIR]T1.COB;1
       P2.
.......^
%COBOL-I-UNREACH, code can never be executed at label P2
at line number 9 in file DISK$YOURDISK:[TESTDIR]T1.COB;1
```

For the same program, the VAX COBOL compiler produces no messages even though the compiler does detect both the unreachable label *and* the unreachable contained program.

Use the /OPTIMIZE qualifier to direct the DEC COBOL compiler to do the uncalled routine analysis. The compiler performs the unreachable code analysis for the default (lowest) level of optimization.

This difference from VAX COBOL can help you when debugging a program. Because these messages are informational, the compiler produces an object file, which you can link and execute. However, these messages may point out otherwise undetected logic errors (that is, the structure of the program is probably not what you intended).

A.3.2.11 COPY and REPLACE Statements

The DEC COBOL compiler produces different output when listing annotations for the COPY statement in COBOL programs.

The two following examples illustrate the difference in the position of the listing annotations, represented by the letter L, in a COBOL program using the VAX COBOL compiler and the DEC COBOL compiler.

VAX COBOL source file:

```
   1        IDENTIFICATION DIVISION.
   2        PROGRAM-ID. DCOP1B.
   3       *
   4       *    This program tests the copy library file.
   5       *    with a comment in the middle of it.
   6       *    It should not produce any diagnostics.
   7            COPY
   8       *    this is the comment in the middle
   9                    LCOP1A.
  10L       ENVIRONMENT DIVISION.
  11L       INPUT-OUTPUT SECTION.
  12L       FILE-CONTROL.
  13L       SELECT FILE-1
  14L           ASSIGN TO "FILE1.TMP".
  15        DATA DIVISION.
  16        FILE SECTION.
  17        FD  FILE-1.
  18        01  FILE1-REC       PIC X.
  19        WORKING-STORAGE SECTION.
  20        PROCEDURE DIVISION.
  21        PE. DISPLAY "***END***"
  22            STOP RUN.
```

DEC COBOL source file:

```
              1 IDENTIFICATION DIVISION.
              2 PROGRAM-ID. DCOP1B.
              3 *
              4 *       This program tests the copy library file.
              5 *       with a comment in the middle of it.
              6 *       It should not produce any diagnostics.
              7         COPY
              8 *       this is the comment in the middle
              9                 LCOP1A.
L            10 ENVIRONMENT DIVISION.
L            11 INPUT-OUTPUT SECTION.
L            12 FILE-CONTROL.
L            13 SELECT FILE-1
L            14         ASSIGN TO "FILE1.TMP".
             15 DATA DIVISION.
             16 FILE SECTION.
             17 FD      FILE-1.
             18 01      FILE1-REC       PIC X.
             19 WORKING-STORAGE SECTION.
             20 PROCEDURE DIVISION.
             21 PE.     DISPLAY "***END***"
             22         STOP RUN.
```

The DEC COBOL compiler also produces different output when listing a COBOL program with multiple COPY statements on a single line, as shown in the next two examples. When the compiler issues a message on a replaced line, the message pointer calls out the original text, not the replacement text.

VAX COBOL source file:

```
    1        IDENTIFICATION DIVISION.
    2        PROGRAM-ID. DCOP1J.
    3       *
    4       *    Tests copy with three copy statements on 1 line.
    5       *
    6        ENVIRONMENT DIVISION.
    7        DATA DIVISION.
    8        PROCEDURE DIVISION.
    9        THE.
   10            COPY LCOP1J.
   11L           DISPLAY "POIUYTREWQ".
   12C                        COPY LCOP1J.
   13L           DISPLAY "POIUYTREWQ".
   14C                                     COPY LCOP1J.
   15L           DISPLAY "POIUYTREWQ".
   16            STOP RUN.
```

DEC COBOL source file:

```
              1 IDENTIFICATION DIVISION.
              2 PROGRAM-ID. DCOP1J.
              3 *
              4 *       Tests copy with three copy statements on 1 line.
              5 *
              6 ENVIRONMENT DIVISION.
              7 DATA DIVISION.
              8 PROCEDURE DIVISION.
              9 THE.
             10         COPY LCOP1J. COPY LCOP1J. COPY LCOP1J.
L            11         DISPLAY "POIUYTREWQ".
L            12         DISPLAY "POIUYTREWQ".
L            13         DISPLAY "POIUYTREWQ".
             14         STOP RUN.
```

The diagnostics for the COBOL source statements REPLACE and DATE-COMPILED result in compiler listings that contain multiple instances of the source line.

For a REPLACE statement listing in a DEC COBOL program, if the compiler issues a message on the replacement text, the compiler message corresponds to the original text in the program. In a VAX COBOL program, however, the compiler message corresponds to the replacement text.

The compiler listing for a DEC COBOL program and a VAX COBOL program differs when a COPY statement inserts text in the middle of a line as shown in the following two examples.

DEC COBOL source file:

```
   -----------------------------------------------------------
     13 P0.     MOVE COPY LCOP5D. TO ALPHA.
  L  14              "O"
```

VAX COBOL source file:

```
-----------------------------------------------------------
13         P0. MOVE COPY LCOP5D.
14L                 "O"
15C                              TO ALPHA.
```

LCOP5D.LIB contains "O". The DEC COBOL compiler keeps the same line and inserts the COPY file contents below the source line. The VAX COBOL compiler splits the original source line into parts.

For the REPLACE and COPY REPLACING statements, program listing line numbers differ between DEC COBOL and VAX COBOL. For DEC COBOL, the line number for the replacement line corresponds to its line number in the original source text, while subsequent line numbers differ. The VAX COBOL compiler arranges the line numbers consecutively.

The following source program can result in listings with different ending line numbers, depending on whether you compile it with the DEC COBOL or the VAX COBOL compiler.

Source file:

```
        REPLACE ==A VERY LONG STATEMENT== by ==EXIT PROGRAM==.
        A
        VERY
        LONG
        STATEMENT.
        DISPLAY "To REPLACE or not to REPLACE".
```

DEC COBOL version:

```
-----------------------------------------------------------------
        1 REPLACE ==A VERY LONG STATEMENT== by ==EXIT PROGRAM==.
        2 EXIT PROGRAM.
        6 DISPLAY "To REPLACE or not to REPLACE".
```

VAX COBOL version:

```
-----------------------------------------------------------------
        1 REPLACE ==A VERY LONG STATEMENT== by ==EXIT PROGRAM==.
        2 EXIT PROGRAM.
        3 DISPLAY "To REPLACE or not to REPLACE".
```

A.3.2.12 MOVE Statement

Unsigned computational fields can hold larger values than signed computational fields. In accordance with the ANSI COBOL Standard, the values for unsigned items should always be treated as positive. VAX COBOL, however, treats unsigned items as signed, while DEC COBOL treats them as positive. Therefore, in some rare cases, a mixture of unsigned and signed data items in MOVE or arithmetic statements can produce different results between VAX COBOL and DEC COBOL.

The following sample program produces different results for VAX COBOL and DEC COBOL.

Source file:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SHOW-DIFF.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 A2       PIC 99    COMP.
01 B1       PIC S9(5) COMP.
01 B2       PIC 9(5)  COMP.
PROCEDURE DIVISION.
TEST-1.
    MOVE 65535 TO A2.
    MOVE A2 TO B1.
    DISPLAY B1 WITH CONVERSION.
    MOVE A2 TO B2.
    DISPLAY B2 WITH CONVERSION.
    STOP RUN.
```

VAX COBOL results:

```
B1 =  -1
B2 =  -1
```

DEC COBOL results:

```
B1 =  65535
B2 =  65535
```

A.3.2.13 ACCEPT and DISPLAY Statements

When you use any extended feature of ACCEPT or DISPLAY within your program, the DEC COBOL compiler uses the DEC SMG (Screen Manager). The visible differences in behavior between DEC COBOL and VAX COBOL are as follows:

- When you run your program, the screen is automatically erased when it encounters the first ACCEPT or DISPLAY statement.
- Because the DEC SMG manages terminal I-O use with extended ACCEPT and DISPLAY statements as screen entities rather than as line by line I-O, you may not be able to redisplay information that appears to have scrolled off the screen by using the DECterm scroll bar.
- The DCL RECALL command is not supported during screen accepts.
- Escape sequence processing is limited to the use of an escape sequence that occupies the leftmost positions of a DISPLAY string. (Sample programs are located in the *DEC COBOL User Manual*.)
- When you mix ANSI ACCEPT statements and extended ACCEPT statements in a program, the editing keys used by the extended ACCEPT statements will also be used by the ANSI ACCEPT statements. (See the *DEC COBOL User Manual* for a complete list of editing keys.)

A.3.2.14 LINAGE Statement

The DEC COBOL and VAX COBOL compilers exhibit different behavior when handling large values for the LINAGE statement. If the line count for the ADVANCING clause of the WRITE statement is larger than 127, DEC COBOL advances one line. VAX COBOL results are undefined.

A.3.2.15 File Status Differences

The DEC COBOL and VAX COBOL compilers report different file status codes when you open a file in EXTEND mode and then try to REWRITE it. DEC COBOL reports a 49 (incompatible open mode). VAX COBOL reports an error 43 (no previous READ).

DEC COBOL sets the file status to 46 after a START fails. VAX COBOL does not produce these results.

A.3.2.16 System Return Codes

The following example illustrates an illegal coding practice that exhibits a certain behavior on OpenVMS VAX systems but that does not produce the same behavior on OpenVMS AXP systems. This difference in behavior points to an architectural difference in the register sets between the VAX and Alpha AXP architectures. Specifically, the difference in behavior on the AXP system is due to the separate set of registers used for floating-point data types.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BADCODING.
ENVIRONMENT DIVISION.

DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.

    01  FIELDS-NEEDED.
        05 CYCLE-LOGICAL      PIC X(14) VALUE 'A_LOGICAL_NAME'.

    01  EDIT-PARM.
        05  EDIT-YR            PIC X(4).
        05  EDIT-MO            PIC XX.

    01  CMR-RETURN-CODE        COMP-1 VALUE 0.

LINKAGE SECTION.

       01 PARM-REC.
          05 CYCLE-PARM         PIC X(6).
          05 RETURN-CODE        COMP-1 VALUE 0.

PROCEDURE DIVISION USING PARM-REC GIVING CMR-RETURN-CODE.

P0-CONTROL.

    CALL 'LIB$SYS_TRNLOG' USING BY DESCRIPTOR CYCLE-LOGICAL,
                                OMITTED,
                                BY DESCRIPTOR CYCLE-PARM
                                GIVING RETURN-CODE.

    IF  RETURN-CODE  GREATER 0
      THEN
        MOVE RETURN-CODE TO CMR-RETURN-CODE
        GO TO P0-EXIT.

    MOVE  CYCLE-PARM  TO  EDIT-PARM.

    IF  EDIT-YR  NOT  NUMERIC
      THEN
        MOVE  4  TO CMR-RETURN-CODE, RETURN-CODE.

    IF  EDIT-MO  NOT NUMERIC
      THEN
        MOVE  4  TO  CMR-RETURN-CODE, RETURN-CODE.
    IF  CMR-RETURN-CODE GREATER 0
                 OR
        RETURN-CODE GREATER 0
      THEN
        DISPLAY "****************************"
        DISPLAY "** BADCODING.COB  **"
        DISPLAY "** A_LOGICAL_NAME> ", CYCLE-PARM, "   **"
        DISPLAY "****************************".

P0-EXIT.

    EXIT PROGRAM.
```

In the sample program, the programmer incorrectly defined the return value for a system service call to be F_floating when it should have been binary (COMP). The programmer was depending on the following VAX behavior: in the VAX architecture, all return values from routines are returned in register R0. The VAX architecture has no separate integer and floating-point registers. The Alpha AXP architecture defines separate register sets for floating-point and binary data. In particular, routines that return floating-point values return them in register F0; routines that return binary values return them in register R0.

The DEC COBOL compiler has no method for determining what data type an external routine may return. You must specify the correct data type for the GIVING-VALUE item in the CALL statement. On OpenVMS AXP systems, the generated code is testing F0 instead of R0 because of the different set of registers used for floating-point data items.

In the sample program, the value in F0 is completely random in this code sequence. In some cases, this coding practice may produce the expected behavior, but in most cases it will not.

A.3.2.17 Storage Differences for Double-Precision Data Items

The difference in storage of D_floating items between the VAX and Alpha AXP architectures produces slightly different answers when validating execution results. The magnitude of the difference depends upon how many D-float computations and stores the compiler performed before outputing the final answer. This behavior difference may cause some difficulty if you attempt to validate output generated by your program running on OpenVMS AXP systems against output generated by OpenVMS VAX systems where they output COMP-2 data to a file.

For information about storage for floating-point data types, see the *Alpha Architecture Handbook*.

A.3.2.18 RMS Special Registers

The DEC COBOL run-time system checks some I-O error situations before attempting the RMS operation. VAX COBOL does the RMS calls without doing any checking, resulting in different values for RMS special registers. When the DEC COBOL run-time system does not attempt an RMS operation, the register value retains its previous value.

For example, in the case of a file that was not successfully opened, any DEC COBOL record operation (READ, WRITE, START, DELETE, REWRITE, or UNLOCK) will fail without invoking RMS.

A.4 Compatibility of DEC Fortran for OpenVMS AXP with VAX FORTRAN

This section discusses the compatibility between DEC Fortran for OpenVMS AXP systems and VAX FORTRAN in the following areas:

- Language features (Section A.4.1)
- Command line qualifiers (Section A.4.2)
- Interoperability with translated shared images (Section A.4.3)
- Porting VAX FORTRAN data (Section A.4.4)

A.4.1 Language Features

DEC Fortran includes ANSI FORTRAN–77 standard features, as well as the VAX FORTRAN extensions to the FORTRAN–77 standard, 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, COMPLEX*8, COMPLEX*16
- IMPLICIT NONE statement
- INCLUDE statement
- NAMELIST I/O
- Names up to 31 characters including use of 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
- Other language elements identified in the *DEC Fortran Language Reference Manual*

For detailed information about extensions, see the *DEC Fortran Language Reference Manual*, which visually shows extensions of the FORTRAN–77 standard.

The remainder of this section summarizes language features specific to VAX FORTRAN and DEC Fortran, language features that are shared but interpreted differently in each language, DEC Fortran restrictions that do not apply to VAX FORTRAN, and data porting considerations.

For complete details about language features, see the *DEC Fortran Language Reference Manual*.

A.4.1.1 Language Features Specific to DEC Fortran

The following language features are available in DEC Fortran but are not supported in VAX FORTRAN Version 5.0:

- Quotation marks (") as delimiters for character constants. This can be disabled by specifying the /VMS qualifier.
- The AUTOMATIC and STATIC statements
- Recursion
- Naturally aligned or packed boundaries for fields of records and items in COMMON blocks
- The POINTER statement data type
- The INTEGER*1, INTEGER*8, and LOGICAL*8 data types
- Support for floating-point S_floating and T_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 AXP systems, see the *Alpha Architecture Reference Manual*.
- LIB$ESTABLISH and LIB$REVERT are provided as intrinsic functions for compatibility with VAX FORTRAN condition handling.
- Bit constants of the form ′0..1′B and B′0..1′.
- The MIL-STD 1753 syntax for octal constants (O′0..7′) and hexadecimal constants (X′0..F′ or Z′0..F′).
- 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 DEC Fortran.
- Warning message “feature not available on this platform” provided for platform-specific features not supported on AXP systems.
- Case-sensitive names
- I/O unit numbers can be any nonnegative integer in DEC Fortran. In VAX FORTRAN, the values for I/O unit numbers can range from 0 to 99.

For an explanation of DEC Fortran language features, see the *DEC Fortran Language Reference Manual*.

A.4.1.2 Language Features Specific to VAX FORTRAN

The following language features are available in VAX FORTRAN but are not supported in DEC 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 DICTIONARY statement (Common Data Dictionary support is not available in the first release of DEC Fortran).
- 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 DEC Fortran does support the DEFINE FILE statement.)

  In place of ERRSET and ERRTST, VMS condition handling might be used. Note that DEC Fortran 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 language features are available in VAX FORTRAN but are not supported in DEC Fortran because of differences between the Alpha AXP architecture and the VAX architecture:

- Certain FORSYSDEF symbol definition modules may be specific to the VAX or Alpha AXP architecture.
- Precise exception control

  The handling of certain exceptions differs between OpenVMS VAX and OpenVMS AXP systems.
- The REAL*16 (H_floating) data type and the REAL*16 Q intrinsic functions
- VAX support for D_floating

  Because the Alpha AXP instruction set does not support the D_floating REAL*8 format, D_floating data is converted to G_floating by software during computations and then converted back to D_floating format. Thus, there will be differences in D_floating arithmetic between VAX and AXP systems.

  For optimal performance on AXP systems, consider using REAL*8 data in VAX G_floating or IEEE T_floating format, perhaps using the /FLOAT qualifier to specify the format. To create a DEC Fortran application program to convert D_floating data to G_floating or T_floating format, use the file conversion methods described in the *DEC Fortran Language Reference Manual*.

- Vectorization capabilities

  Vectorization associated with the VAX FORTRAN High-Performance Option (HPO), including /VECTOR and its related qualifiers, and the CDEC$ INIT_DEP_FWD directive are not supported. The Alpha AXP processor provides pipelining and other features that resemble vectorization capabilities.

A.4.1.3 Interpretation Differences

The following language features are interpreted differently between VAX FORTRAN and DEC Fortran:

- Octal notation for integer constants
- Random number generator (RAN)

  The RAN function generates a different pattern of numbers in DEC Fortran than in VAX FORTRAN for the same random seed. (The RAN and RANDU functions are provided for VAX FORTRAN compatibility.)
- Hollerith constants in formatted I/O statements

  VAX FORTRAN and DEC 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 VAX FORTRAN, the value obtained by the READ statement is the output of the WRITE statement (FMT2 is ignored). However, in DEC 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.)

A.4.1.4 DEC Fortran Restrictions

Certain VAX FORTRAN features have restricted use or are not available in DEC 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 will be 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. (VAX FORTRAN passed 'A' by reference if the dummy argument was numeric.)

- 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.

A.4.2 Command Line Qualifiers

This section summarizes the differences between DEC Fortran and VAX FORTRAN command line qualifiers.

For complete details about the DEC Fortran compilation command and options, see the *DEC Fortran User Manual for OpenVMS AXP Systems*. For complete details about the VAX FORTRAN compilation command and options, see the *** *WARNING: OBSOLETE. Use DEC_FORT_VMS_VAX_UM.* ***.

To initiate compilation on either VAX or AXP systems, use the FORTRAN command.

While some compiler qualifiers are specific to each language, DEC Fortran and VAX FORTRAN share many qualifiers.

A.4.2.1 Shared Qualifiers

Table A–10 lists the compiler qualifiers that are shared by DEC Fortran and VAX FORTRAN. For detailed information about DEC Fortran qualifiers, see the *DEC Fortran Language Reference Manual*.

Table A–10 Qualifiers Shared by DEC Fortran and VAX FORTRAN

Qualifier	Comments
/ASSUME	/ASSUME was not available in VAX FORTRAN Version 5.0, but /ASSUME=([NO]ACCURACY_SENSITIVE,[NO]DUMMY_ALIAS) are now available in VAX FORTRAN HPO.
	Certain keyword values are specific to DEC Fortran.
/ANALYSIS_DATA	Equivalent.
/CHECK	All VAX FORTRAN /CHECK= keywords are available in DEC Fortran. The DEC Fortran /WARNINGS=ALIGNMENT qualifier and the VAX FORTRAN HPO /CHECK=ALIGNMENT qualifier are equivalent.
/CROSS_REFERENCE	Equivalent.
/DEBUG	All VAX FORTRAN /DEBUG= keywords are available in DEC Fortran.
/D_LINES	Equivalent.
/DIAGNOSTICS	Equivalent.
/DML	Equivalent.
/EXTEND_SOURCE	Equivalent.

(continued on next page)

Table A–10 (Cont.) Qualifiers Shared by DEC Fortran and VAX FORTRAN

Qualifier	Comments
/F77	Equivalent.
/G_FLOATING	Although DEC Fortran supports /G_FLOATING, use the DEC Fortran /FLOAT qualifier instead. Note that differences exist for D_floating computations on VAX and AXP systems, as described in the description of /FLOAT in Section A.4.2.2.
/I4	Equivalent. With DEC Fortran, you can use the /INTEGER_SIZE qualifier to specify the size of INTEGER declarations.
/LIBRARY	Equivalent.
/LIST	Equivalent.
/MACHINE_CODE	Equivalent.
/OBJECT	Equivalent.
/OPTIMIZE	Equivalent, although DEC Fortran also supports the use of optimization levels such as /OPTIMIZE=LEVEL=1 for only local optimizations (see Section A.4.2.2). Actual compiler optimization techniques may differ.
/SHOW	Most VAX FORTRAN /SHOW= keywords are available in DEC Fortran.
/STANDARD	All VAX FORTRAN /STANDARD= keywords are available in DEC Fortran.
/WARNINGS	Most /WARNINGS= keywords are available; however, certain keyword values are specific to DEC Fortran.

Consider using the DEC Fortran /VMS qualifier (the default) when porting VAX FORTRAN source programs.

A.4.2.2 Qualifiers Specific to DEC Fortran

Table A–11 lists DEC Fortran compiler qualifiers that have no equivalent VAX FORTRAN options and are not supported in VAX FORTRAN Version 5.0.

Table A–11 DEC Fortran Qualifiers Not in VAX FORTRAN

Qualifier	Description
/ALIGN	Controls alignment of record structures and common blocks. (VAX FORTRAN handles the default alignment for common blocks and also recognizes a CDEC$ directive.)
/ASSUME	Certain keywords are not provided by VAX FORTRAN Version 5.0, including BIG_ENDIAN, CRAY, IBM, LITTLE_ENDIAN, RECURSIVE, VAXD, and VAXG.
/FLOAT	Controls the format used for floating-point data (REAL or COMPLEX), allowing use of VAX G_floating, VAX D_floating, or IEEE (S_floating and T_floating) floating-point data.
/INTEGER_SIZE	Controls the size of INTEGER declarations.
/NAMES	Controls whether external names are converted to uppercase, lowercase, or left as is.

(continued on next page)

Table A–11 (Cont.) DEC Fortran Qualifiers Not in VAX FORTRAN

Qualifier	Description
/OPTIMIZE=LEVEL=*n*	Controls the level of optimization between /NOOPTIMIZE and /OPTIMIZE (/OPTIMIZE=LEVEL=4). VAX FORTRAN (and DEC Fortran) supports /OPTIMIZE and /NOOPTIMIZE.
/POINTER_SIZE	Controls the size (addressable range) of pointer data.
/RECURSIVE	Allocates local data on the run-time process stack and prepages procedures for possible recursive execution.
/VMS	Requests that DEC Fortran use certain VAX FORTRAN conventions.
/WARNING=(ALIGNMENTS, TRUNCATED_SOURCE)	Requests that warning messages appear for any data that is not naturally aligned and any source lines that are truncated. These keywords are available in VAX FORTRAN HPO.

A.4.2.3 Qualifiers Specific to VAX FORTRAN

This section summarizes VAX FORTRAN compiler options that have no equivalent DEC Fortran options.

Table A–12 lists compilation options that are specific to VAX FORTRAN Version 5.0.

Table A–12 VAX FORTRAN Options Not in DEC Fortran

VAX FORTRAN Qualifier	Description
/BLAS=(INLINE,MAPPED)	Specifies whether VAX FORTRAN recognizes and inlines or maps the Basic Linear Algebra Subroutines (BLAS). Available only for the VAX FORTRAN High Performance Option (HPO).
/CHECK=ASSERTIONS	Enables or disables assertion checking. Available only for VAX FORTRAN HPO.
/CONTINUATIONS=*n*	Specifies the number of continuation lines allowed in a statement. DEC Fortran allows up to 99 continuation lines.
/DESIGN=[NO]COMMENTS /DESIGN=[NO]PLACEHOLDERS	Analyzes program for design information.
/DIRECTIVES=DEPENDENCE	Specifies whether specified compiler directives are used at compilation. Available only for VAX FORTRAN HPO.
/MATH_LIBRARY=(FAST or ACCURATE)	Controls the selection of math library routines used to implement certain mathematical intrinsic functions. /MATH_LIBRARY also affects vectorized references to the exponentiation operator ** for real data types. Available only in VAX FORTRAN HPO.
/PARALLEL=(MANUAL or AUTOMATIC)	Supports parallel processing.

(continued on next page)

Table A–12 (Cont.) VAX FORTRAN Options Not in DEC Fortran

VAX FORTRAN Qualifier	Description
/SHOW=(DATA_DEPENDENCIES,DICTIONARY,LOOPS)	Control whether the listing file includes: • Diagnostics about loops that are ineligible for dependence analysis and data dependencies that inhibit vectorization or autodecomposition (DATA_DEPENDENCIES) • Source lines from included Common Data Dictionary records (DICTIONARY) • Reports about loop structures after compilation (LOOPS) The keywords DATA_DEPENDENCIES and LOOPS are available only for VAX FORTRAN HPO.
/VECTOR	Requests vector processing. Available only with VAX FORTRAN HPO.
/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 for VAX FORTRAN HPO.

All CPAR$ directives and certain CDEC$ directives associated with directed (manual) decomposition and their associated qualifiers or keywords are also specific to VAX FORTRAN, as described in the *DEC Fortran Language Reference Manual*.

For details about the VAX FORTRAN compilation commands and options, see the *** *WARNING: OBSOLETE. Use DEC_FORT_VMS_VAX_UM.* ***.

A.4.3 Interoperability with Translated Shared Images

Using DEC 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 command line, specify the /TIE qualifier.
- On the LINK command line, specify the /NONATIVE_ONLY qualifier (default).

The created executable image contains code that allows the resulting executable image to interoperate with shared images, including allowing the VAX FORTRAN RTL (FORRTL) to work with the DEC Fortran RTL (DEC$FORTRTL). The native (DEC Fortran RTL) and translated (VAX FORTRAN 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.

A.4.4 Porting VAX FORTRAN Data

Record types are identical for VAX FORTRAN and DEC 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 VAX FORTRAN programs (VAX hardware) store REAL*4 or COMPLEX*8 data in F_floating format, REAL*8 or COMPLEX*16 data in either D_floating or G_floating format, and REAL*16 data in H_floating format. DEC Fortran programs (running on Alpha AXP hardware) store REAL*4, REAL*8, COMPLEX*8, and COMPLEX*16 data in one of the formats shown in Table A–13.

Table A–13 Floating-Point Data on VAX and AXP Systems

Data Declaration	VAX Formats	AXP Formats
REAL*4 and COMPLEX*8	VAX F_floating format	IEEE S_floating or VAX F_floating 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 and COMPLEX*32	VAX H_floating	Not supported by DEC Fortran. Requires conversion, perhaps using the RTL routine CVT$CONVERT_FLOAT.

[1]On AXP systems, the use of VAX D_floating format involving many computations is not recommended. Consider converting D_floating format to IEEE T_floating (or VAX G_floating) format in a conversion program that uses the DEC Fortran conversion routines.

You may not be able to convert the VAX H_floating (REAL*16) data outside the range of VAX D_floating or G_floating format (REAL*8). One option is to write a conversion application to convert the files to VAX D_floating or G_floating format (REAL*8) and then use the DEC Fortran conversion routines to convert the data to IEEE T_floating format.

A.5 Compatibility of DEC Pascal for OpenVMS AXP Systems with VAX Pascal

This section compares DEC Pascal to other Digital Pascal compilers and lists the differences between DEC Pascal on VAX and AXP systems. For a complete description of these features, see the *DEC Pascal Language Reference Manual*.

A.5.1 New Features of DEC Pascal

Table A–14 lists features not previously supplied in VAX Pascal.

Table A–14 New Features of DEC Pascal

Feature	Description
Support for OpenVMS systems	Including all the data types available on the OpenVMS platforms.
Redefinable values for predeclared constants	Values for MAXINT, MAXUNSIGNED, MAXREAL, MINREAL, ESPREAL are defined by the platform and the compiler switches for specifying the integer size and floating-point format.

(continued on next page)

Table A–14 (Cont.) New Features of DEC Pascal

Feature	Description
An optional quoted parameter to the COMMON, EXTERNAL, GLOBAL, PSECT, WEAK_EXTERNAL, and WEAK_GLOBAL attributes	Allows you to pass an unmodified identifier to the linker.
Double-quoted strings	DEC Pascal now accepts the double-quote characters as string and character delimiters.
Embedded string values	Inside of double-quoted strings, DEC Pascal now supports constant characters specified with a backslash as in the C programming language, such as “"\ n"” for the linefeed character.
Additional data types and values	DEC Pascal now supports these data types: ALFA, CARDINAL, CARDINAL16, CARDINAL32, INTEGER16, INTEGER32, INTEGER64, INTSET, POINTER, UNIV_PTR, UNSIGNED16, UNSIGNED32, and UNSIGNED64.
Assignment of UNSIGNED values to INTEGER variables	DEC Pascal now allows UNSIGNED values to be assignment-compatible with INTEGER variables and array indices.
Assignment of string values into unpacked arrays of characters	DEC Pascal now allows ARRAY of CHAR variables to be treated as fixed-length character strings.
Additional statements	DEC Pascal now supports these statements: BREAK, CONTINUE, EXIT, NEXT, and RETURN.
Additional predeclared routines	DEC Pascal now supports these functions and procedures: ADDR, ARGC, ARGV, ASSERT, BITAND, BITNOT, BITOR, BITXOR, HBOUND, LBOUND, FIRST, FIRSTOF, LAST, LASTOF, IN_RANGE, LSHIFT, RSHIFT, LSHFT, RSHFT, MESSAGE, NULL, RANDOM, SEED, REMOVE, SIZEOF, SYSCLOCK, and WALLCLOCK.
Optional second parameter to RESET, REWRITE, and EXTEND	DEC Pascal now accepts a second parameter that is a literal string expression for the file name to be associated with the file variable.
Compiler command switches	DEC Pascal now includes switches that allow you to specify the storage and alignment allocation for data types. You can also specify the level of optimization with a switch. On AXP systems, an option controls the default meaning of the REAL and DOUBLE data types. Arguments to the usage switch enable messages relating to alignment, alignment compatibility on different platforms, and features that are not available on a specified platform.

A.5.2 Modifying Default Alignment Rules for Record Fields

DEC Pascal allows you to override field alignment and position with the POS, ALIGNED, and DATA attributes and the data compiler switch.

A.5.3 Recommended Use of Predeclared Identifiers

Although for backward compatibility DEC Pascal compiles programs that include the predeclared identifiers listed in Table A–15, Digital recommends that you use the listed replacements.

Table A–15 Recommended Use of Predeclared Identifiers

Identifier	Recommended Usage
ADDR	Use the ADDRESS function
ALFA	Equivalent to TYPE ALFA = PACKED ARRAY [1..10]OF CHAR
BITAND	Equivalent to the UAND statement
BITNOT	Equivalent to the UNOT statement
BITOR	Equivalent to the UOR statement
BITXOR	Equivalent to the UXOR statement
EXIT	Equivalent to the BREAK statement
FIRST, FIRSTOF	Equivalent to the LOWER function
HBOUND	Equivalent to the UPPER function
IN_RANGE	Useful only when subrange checking is disabled. IN_RANGE(X) is equivalent to (X≥LOWER(X))AND(X≤UPPER(X)).
INTSET	Equivalent to TYPE INTSET = SET OF 0 .. 255;
LAST, LASTOF	Equivalent to the UPPER function
LBOUND	Equivalent to the LOWER function
LSHFT	Equivalent to the LSHIFT function
MESSAGE	Equivalent to WRITELN(ERR,expression)
NEXT	Equivalent to the CONTINUE statement
NULL	Equivalent to the empty statement
REMOVE	Equivalent to the DELETE_FILE procedure
RSHFT	Equivalent to the RSHIFT function
SIZEOF	Equivalent to the SIZE function
STLIMIT	Compiles but does not return an error
UNIV_PTR	Equivalent to TYPE UNIV_PTR = POINTER;

A.5.4 Platform-Dependent Features

DEC Pascal can use an environment file only on the same platform (the combination of operating system and hardware) on which it was compiled.

In addition, the following lists features of DEC Pascal supplied only on VAX systems:

- QUADRUPLE data type
- H_floating-point data type
- VAX Pascal Version 1.0 dynamic arrays
- MFPR and MTPR predeclared routines
- [OVERLAID] attribute

- Table of contents in listing
- Optimize attribute on routines

The following lists the features of DEC Pascal that are supplied only on AXP systems:

- Abbreviations when reading enumerated data types
- Indexed file organization
- Relative file organization

A.5.5 Obsolete Features

This section describes features that are supported, but not recommended, by Digital. They are provided only for compatibility with other Digital Pascal compilers.

A.5.5.1 /OLD_VERSION Qualifier

The /OLD_VERSION qualifier directed the compiler to resolve differences between VAX Pascal Version 1.0 and subsequent versions by using the VAX Pascal Version 1.0 definition of the language. The qualifier is provided so that existing programs continue to work.

A.5.5.2 /G_FLOATING Qualifier

The /G_FLOATING qualifier directs the compiler to use the G_floating representation and instructions for values of type DOUBLE. The [[NO]G_FLOATING] attribute can be specified on both OpenVMS VAX and OpenVMS AXP systems.

If the use of the /G_FLOATING qualifier conflicts with a double-precision attribute specified in the source program or module, an error occurs. Routines and compilation units between which double-precision quantities are passed should not mix floating-point formats. Not all OpenVMS VAX processors support the G_floating data types.

See also the description of the /FLOAT qualifier, which is the preferred method for specifying the floating-point format to the compiler. The /FLOAT qualifier also allows you to select the IEEE floating-point format, which is supported only on AXP systems.

A.5.5.3 OVERLAID Attribute

The OVERLAID attribute indicated how storage should be allocated for variables declared within a compilation unit. If you specify OVERLAID on a compilation unit, the variables declared at program or module level (unless they have the STATIC or PSECT attribute) overlay the storage of static variables in all other overlaid compilation units.

This attribute is intended for use only with programs that use the decommitted separate compilation facility provided by VAX Pascal Version 1.0.

Index