Creating an OpenVMS AXP Step 2 Device Driver from an OpenVMS VAX Device Driver

Order Number: AA–Q28UA–TE

March 1994

This manual describes how to convert an OpenVMS VAX device driver, written in VAX MACRO, to an OpenVMS AXP Step 2 driver, also written in VAX MACRO.

Revision/Update Information: This is a new manual.

Software Version: OpenVMS AXP Version 6.1

Digital Equipment Corporation
Maynard, Massachusetts

March 1994

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

This document is available on CD–ROM.

ZK6362

This document was prepared using VAX DOCUMENT Version 2.1.

Send Us Your Comments

We welcome your comments on this or any other OpenVMS manual. If you have suggestions for improving a particular section or find any errors, please indicate the title, order number, chapter, section, and page number (if available). We also welcome more general comments. Your input is valuable in improving future releases of our documentation.

You can send comments to us in the following ways:

- Internet electronic mail: `OPENVMSDOC@ZKO.MTS.DEC.COM`
- Fax: `603-881-0120 Attn: OpenVMS Documentation, ZKO3-4/U08`
- A completed Reader's Comments form (postage paid, if mailed in the United States), or a letter, via the postal service. Two Reader's Comments forms are located at the back of each printed OpenVMS manual. Please send letters and forms to:


```
Digital Equipment Corporation
Information Design and Consulting
OpenVMS Documentation
110 Spit Brook Road, ZKO3-4/U08
Nashua, NH  03062-2698
USA
```


You may also use an online questionnaire to give us feedback. Print or edit the online file SYS$HELP:OPENVMSDOC_SURVEY.TXT. Send the completed online file by electronic mail to our Internet address, or send the completed hardcopy survey by fax or through the postal service.

Thank you.

Contents

4 Allocating Map Registers and Other Counted Resources

5 Synchronization Requirements for OpenVMS AXP Device Drivers

6 Conversion Guidelines

7 Handling Complex Conversions Situations

Index

Examples

Figures

Tables

Preface

This manual describes how to convert an OpenVMS VAX device driver to an OpenVMS AXP Step 2 device driver. It explains how you must change OpenVMS VAX driver code to prepare the driver to be compiled, linked, loaded, and run as a Step 2 driver. This manual highlights specific changes that you must make to driver routines and tables, and indicates how OpenVMS VAX data structures, macros, and executive routines upon which drivers rely have been modified for the OpenVMS AXP operating system.

Intended Audience

Creating an OpenVMS AXP Step 2 Device Driver from an OpenVMS VAX Device Driver is intended for software engineers who must prepare an OpenVMS VAX device driver to run on the OpenVMS AXP operating system, Version 6.1.

This manual assumes that its reader is familiar with the components of OpenVMS VAX device drivers. It also relies on a familiarity with the software interfaces within the OpenVMS operating system that support device drivers.

Document Structure

This manual contains the following sections:

- Chapter 1 presents an overview of the new OpenVMS AXP device drivers interfaces.
- Chapter 2 describes how to access device interface registers using hardware I/O mailboxes by means of the controller register access mailbox (CRAM) structure defined by the OpenVMS AXP operating system.
- Chapter 3 discusses the suspension mechanisms OpenVMS AXP device drivers can use, including simple fork semantics and the OpenVMS kernel process services.
- Chapter 4 describes how you request and allocate a counted resource, such as a set of map registers.
- Chapter 5 focuses on the special synchronization needs of OpenVMS AXP device drivers.
- Chapter 6 contains basic guidelines for converting an OpenVMS VAX device driver to an OpenVMS AXP Step 2 device driver.
- Chapter 7 provides tips for converting complex or unusual drivers.

Associated Documents

Creating an OpenVMS AXP Step 2 Device Driver from an OpenVMS VAX Device Driver focuses only on those changes that must be made to an OpenVMS VAX device driver to produce an equivalent Step 2 OpenVMS AXP device driver. For more detailed information about the macros and routines mentioned in this manual, see *OpenVMS AXP Device Support: Reference*. For basic information about the components of OpenVMS device drivers and OpenVMS requirements for them, refer to the following manuals:

- *OpenVMS AXP Device Support: Developer's Guide*
- *OpenVMS AXP Device Support: Reference*

Because this manual only addresses the porting to OpenVMS AXP of VAX MACRO coding practices that are typically found in device drivers, readers who need additional information on porting MACRO code, or a detailed description of the MACRO-32 compiler for OpenVMS AXP, should see *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code*.

Several manuals are available that describe the internals of the OpenVMS AXP operating system and the processes for investigating the types of system failures caused by device drivers. These manuals include:

- *OpenVMS AXP System Dump Analyzer Utility Manual*
- *OpenVMS Delta/XDelta Debugger Manual*
- *OpenVMS for Alpha Platforms: Internals and Data Structures*

Conventions

In this manual, every use of OpenVMS VAX means the OpenVMS VAX operating system, every use of OpenVMS AXP means the OpenVMS AXP operating system, and every use of OpenVMS means both the OpenVMS VAX operating system and the OpenVMS AXP 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.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1, then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	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 options. (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 online versions of the manual.
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, the name of a file protection code, 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

OpenVMS AXP Version 6.1 introduces formal support for user-written device drivers and a new device driver interface known as the **Step 2** driver interface. If you are supplying a driver to run under OpenVMS AXP Version 6.1, it must comply with the OpenVMS AXP Step 2 driver interfaces described in this manual.

Although the Step 2 driver interfaces allow you to write OpenVMS AXP device drivers in high-level languages, an OpenVMS AXP device driver can also be written in VAX MACRO. The guidelines in this manual describe how to convert an OpenVMS VAX driver written in VAX MACRO to an OpenVMS AXP driver written in VAX MACRO.

1.1 Overview of OpenVMS AXP Driver Changes

OpenVMS AXP device drivers differ from OpenVMS VAX device drivers in the following ways:

- You must identify OpenVMS AXP device drivers as Step 2 drivers. See Chapter 6.
- You must explicitly identify driver code and data by using new macros. See Section 6.1.
- An OpenVMS AXP device driver must use multiprocessing synchronization mechanisms, regardless of whether it will operate in an OpenVMS AXP multiprocessing environment. See Section 5.1.
- An OpenVMS AXP device driver should access device control and status registers (CSRs) using the operating system routines described in Chapter 2.
- You must examine existing driver suspension mechanisms (such as fork or fork and wait) to determine whether you need to replace them with the new kernel process services or with the new simple fork mechanism. This decision is made based on whether a driver routine relies on context from a previously called routine on the stack. See Chapter 3.
- The OpenVMS AXP operating system, unlike the OpenVMS VAX operating system, does not manage map registers within fields of the Adapter Control Block (ADP). Rather, it manages map register allocation in the more generic manner described in Chapter 4.
- To produce the object file for an OpenVMS AXP device driver, you must compile the source module or modules with the MACRO-32 compiler for OpenVMS AXP. The compiler relies on the placement of entry point directives for JSB entry points. It also identifies, where possible, coding practices that are illegal on OpenVMS AXP systems (such as coroutine calls and return to caller's caller). See Chapter 6.
- You must declare the entry points of the controller and unit initialization routines using arguments to the DPTAB macro. See Section 6.3.2.

- You must declare the entry point of any interrupt service routine using the new DPT_STORE_ISR macro. See Section 6.4.
- In some cases, changes to driver macros and system routines may require changes to driver code. See *OpenVMS AXP Device Support: Reference* for more information.
- Data structures have been greatly overhauled. Fields have been deleted, expanded, and added. Many field aliases have been removed. Use *OpenVMS AXP Device Support: Reference* while compiling your driver to correct obsolete symbolic offsets. If your driver uses fields that have been removed from the unit control block (UCB) for OpenVMS AXP, Digital recommends using the $DEFINI, $DEF, $DEFEND, and associated macros to create the needed fields in a UCB extension.
- Step 2 drivers are loadable executive images and loaded by the executive loader, which affects how drivers are linked and loaded.
- The driver-loading procedure requires driver controller and unit initialization routines to return a status value in R0. See Section 6.8.

  For more information about linking and loading OpenVMS AXP device drivers, see the *OpenVMS AXP Device Support: Developer's Guide*.
- FDT routines cannot access the $QIO function-dependent parameters by using AP offsets. Instead, you must use the new IRP$L_QIO_Pn cells.
- Drivers must not use floating-point instructions. See Section 6.19 for a full explanation.
- OpenVMS AXP drivers require standard call interfaces for the following driver-supplied routines:
  - Cancel I/O routine
  - Cancel selective routine
  - Channel assign routine
  - Cloned UCB routine
  - Controller initialization routine
  - Function decision table (FDT) routines
  - Interrupt service routine
  - Mount verification routine
  - Register dumping routine
  - Unit delivery routine
  - Unit initialization routine
- Standard call interfaces are optional for the following driver-supplied routines:
  - Alternate start I/O routine
  - Start I/O routine
  - Driver fork routines

- Additional OpenVMS AXP driver changes include the following:
  - Function decision table (FDT) processing does not rely on the RET under JSB mechanism.
  - The layout of the FDT is significantly different.
  - Standard call interfaces are available for most OpenVMS support routines.
  - A small number of OpenVMS support routines with JSB interfaces are no longer available.

For detailed information about these changes, see Chapter 6.

Special guidelines apply to terminal port drivers (see Section 6.17) and drivers for devices with programmable interrupt vectors (see Section 6.18).

1.2 Overview of OpenVMS VAX and OpenVMS AXP Driver Similarities

OpenVMS AXP drivers are similar to OpenVMS VAX drivers in the following ways:

- The overall structure of a device driver is unchanged.
- JSB interfaces continue to be available for most OpenVMS support routines used by drivers.
- Although call interfaces are required for many routines, you can continue to use JSB interfaces for the start I/O to REQCOM code path, OpenVMS support routines, and internal driver routines.

1.3 OpenVMS AXP Driver Routine Naming Conventions

Some OpenVMS AXP driver routine names are different from the OpenVMS VAX routine names. If a routine interface changed because of the AXP architecture, the routine name changed. OpenVMS AXP also includes new call-based system routines. The following naming conventions apply to the new OpenVMS AXP call-based system routines:

- The call-based system routine has a different name than its JSB-based counterpart. If x$y is the name of the JSB-based system routine, its call-based counterpart is named x_STD$y. For example, EXE_STD$FINISHIO is the call-based routine that replaces the JSB-based EXE$FINISHIO.
- If a JSB-replacement macro exists for x$y, it is named CALL_Y.

  For example, you can replace a JSB to EXE$FINISHIO with the CALL_FINISHIO macro. CALL_FINISHIO issues a standard call to EXE_STD$FINISHIO after loading the standard call argument registers from the general registers used in the traditional JSB to EXE$FINISHIO.
- When using the call-based system routine directly, note that its interface may differ from the traditional JSB-based routine.

  Input parameters are usually listed first, specified in the order that corresponds to the register order of the JSB interface input parameters.

  Output parameters are usually listed last, specified in the order that corresponds to the register order of the JSB interface output parameters.

If a register parameter is both an input and an output parameter to the JSB interface, then it contributes both an input parameter and an output parameter to the new call-based interface.

These conventions serve only as guidelines. In some cases, parameters are dropped or the register order rule is waived if an alternate parameter ordering is more natural. All such interface changes are described in *OpenVMS AXP Device Support: Reference*.

1.4 Converting OpenVMS VAX Drivers Written in BLISS

This manual focuses on converting existing OpenVMS VAX device drivers, written in VAX MACRO, to OpenVMS AXP device drivers. However, the call interfaces described are equally available to OpenVMS VAX drivers written in BLISS. To convert an OpenVMS VAX BLISS driver, remove the JSB linkages from routine declarations and verify the specified parameter order for any given routine against that listed in the system routines section of *OpenVMS AXP Device Support: Reference*.

Existing BLISS drivers are likely to have an associated VAX MACRO module that contains the DPTAB, DDTAB, and FUNCTAB declarations, and some routines that were written in VAX MACRO. You must convert these VAX MACRO modules as described in this manual. Alternatively, you can now use new BLISS macros that allow you to code the DPT, DDT, and FDT declarations in BLISS. For more information about these macros, see *OpenVMS AXP Device Support: Reference*.

1.5 Writing OpenVMS AXP Drivers in C

OpenVMS AXP Version 6.1 provides the support necessary to write a device driver in the C programming language. For information about writing OpenVMS AXP device drivers in C or another high-level language, see the *OpenVMS AXP Device Support: Developer's Guide*.

1.6 Using Common Source Code for OpenVMS VAX and OpenVMS AXP Drivers

The OpenVMS AXP Step 2 driver interface has increased the differences between OpenVMS AXP and OpenVMS VAX device drivers. A key difference is that while OpenVMS AXP drivers can be written in the C programming language, there is no formal support for writing OpenVMS VAX device drivers in C. For example, OpenVMS VAX does not provide .h files for internal OpenVMS data structures.

Device driver source files written in MACRO-32 or BLISS can be kept common between OpenVMS AXP and OpenVMS VAX through the use of conditional compilation and user-written macros. The advisability of this approach depends greatly on the nature of the individual driver. It is likely that in future versions of OpenVMS AXP, the I/O subsystem will continue to evolve in directions that will have an impact on device drivers. This could increase the differences between OpenVMS AXP and OpenVMS VAX device drivers and add more complexity to common driver sources. For this reason, a fully common driver source file approach might not be advisable for the long term. However, depending on the individual driver, it may be advisable to divide the driver into a common module and an architecture-specific one. For example, if you were writing a device driver that does disk compression, then the compression algorithm could be isolated into an architecture independent module. You could also avoid operating-system-specific data structures in such common modules with the intent of having some common modules across various types of operating systems; for example, OpenVMS, Windows NT, and OSF.

For more information about writing OpenVMS AXP device drivers in C, see the *OpenVMS AXP Device Support: Developer's Guide*.

2 Accessing Device Interface Registers

A **hardware interface register** is the place where software interfaces with a hardware component. Every hardware component on an OpenVMS AXP system, including CPU and memory, has a set of interface registers.

The portion of a processor's physical address space through which it accesses hardware interface registers is known as its **I/O space**.

In the VAX architecture, a hardware implementation usually defines a physical address boundary between memory space and I/O space. I/O space physical addresses are mapped into the processors' virtual address space and are accessed using VAX load and store instructions (for example, MOV, BIS, and others).

For AXP systems, there are no rules governing how hardware implementations allow access to I/O space. Some AXP platforms allow VAX-style I/O space access. Other platforms provide access to I/O space through **hardware I/O mailboxes**. Some platforms implement both styles of I/O register access.

The challenge presented by the AXP architecture is to create software abstractions that hide the hardware mechanisms for I/O space access from the programmer. These software abstractions contribute to driver portability. The AXP architecture also defines no byte or word length load and store instructions. Because some I/O buses and adapters require byte or word register access granularity for correct adapter operation, AXP system hardware designers invented the following mechanisms that provide byte and word access granularity for I/O adapter register access:

- **Sparse space addressing**, which means the device address space is expanded by a factor of two to allow for inclusion of a byte mask in the write data.
- **Swizzle space addressing**, which means where upper order bits in the processor physical address map to an I/O bus address, while lower order bits are used to implement I/O bus byte enable signals. This causes a large amount of processor physical address space to represent the I/O bus address space.
- **Hardware I/O mailboxes**, which are 64-byte, naturally-aligned, physically-contiguous data structures (defined by the AXP architecture) built in system memory and accessed by special I/O subsystem hardware. Drivers can use hardware I/O mailboxes to deliver commands and write data to the interface registers of a device residing on an I/O bus.

A significant part of I/O bus support in the OpenVMS AXP operating system is to provide standard ways to access I/O device registers. OpenVMS AXP provides a set of data structures and routines that can be used for register access on any system, regardless of the underlying I/O hardware. Bus support provides two ways. One way is the CRAM data structure. The other way is the platform independent access routines IOC$READ_IO and IOC$WRITE_IO.

Note

In register access discussions, the term **control and status register** (CSR) is sometimes used instead of the generic term **interface register**. In this manual, the terms are equivalent.

2.1 Mapping I/O Device Registers

Unlike OpenVMS VAX systems (where the operating system maps registers) before you access device registers on OpenVMS AXP systems, you must map the registers into the processor's virtual address space. OpenVMS AXP provides the IOC$MAP_IO routine, which allows a caller to request mapping based on device characteristics without regard to the platform hardware implementation of I/O space access.

Note

Register mapping is not required on XMI devices on Laser, and IOC$READ_IO and IOC$WRITE_IO are not supported. If you are porting an OpenVMS VAX XMI device driver to an OpenVMS AXP system, you must use CRAMs.

Once your device is mapped, you can access it using a CRAM data structure and associated routines, or the IOC$READ_IO and IOC$WRITE_IO routines.

2.2 Platform Independent I/O Bus Mapping

The platform independent I/O bus mapping routine is called IOC$MAP_IO. This routine maps I/O bus physical address space into an address region accessible by the processor. The caller of this routine can express the mapping request in terms of the bus address space without regard to address swizzling, dense space, sparse space, and so on.

IOC$MAP_IO is supported on PCI, EISA, Turbochannel, and Futurebus+. It is not supported on XMI.

The following new platform independent mapping and access routines exist:

- IOC$MAP_IO
- IOC$READ_IO
- IOC$WRITE_IO
- IOC$UNMAP_IO

The IOC$MAP_IO routine maps I/O bus physical address space into an address region accessible by the processor. The IOC$UNMAP_IO routine is provided to unmap a previously mapped space, returning the IOHANDLE and the PTEs to the system. IOC$READ_IO and IOC$WRITE_IO are platform independent I/O access routines that provide a platform independent way to read and write I/O space without the overhead of CRAM allocation and initialization. These routines require that the I/O space that is to be accessed have been previously mapped by a call to IOC$MAP_IO. For more information about these routines, see *OpenVMS AXP Device Support: Reference*.

2.2.1 Using the IOC$MAP_IO Routine

Drivers that need to use the IOC$MAP_IO routine must call that routine under specific spinlock restrictions. The driver cannot be holding any spinlocks that prohibit IOC$MAP_IO from taking out the MMG spinlock.

Most drivers want to call IOC$MAP_IO immediately after they are loaded. Traditionally, the correct place for a driver to call IOC$MAP_IO would be its controller or unit initialization routine. However, because the controller and unit initialization routines are called at IPL$_POWER, IOC$MAP_IO cannot take out the MMG spinlock in this environment.

The new driver support feature for calling IOC$MAP_IO has two elements. First, the driver may request preallocated space for any number of I/O Handles (the output of IOC$MAP_IO). Second, the driver may name a routine that will be called in an environment suitable for calls to IOC$MAP_IO.

Drivers can specify the number of I/O Handles they need to store using the IOHANDLES parameter on the DPTAB macro. The default parameter value is zero. The maximum permitted value is 65,535.

When the IOHANDLES parameter is zero or one, the driver loader does NOT allocate any additional space for I/O Handles. For these two values, the driver is expected to store the I/O Handle it needs directly in the IDB$Q_CSR field.

When the IOHANDLES parameter is greater than one, an MCJ data structure is allocated. The base address of the MCJ is stored in the low-order longword of IDB$Q_CSR and the IDB$V_MCJ flag is set in IDB$L_FLAGS. MCJ$Q_ENTRIES is the base address in the MCJ of an array of quadword I/O Handle slots. The number of slots in the array is exactly the number specified by the IOHANDLES DPTAB parameter.

Drivers specify a CSR Mapping routine using the CSR_MAPPING parameter on the DDTAB macro. The driver loading procedure calls the CSR_MAPPING routine holding the IOLOCK8 spinlock before it calls the controller or unit initialization routines. In this context, the driver can make all its needed calls to IOC$MAP_IO and other bus support routines with similar calling requirements.

Note

The CSR mapping routine is not called on power fail recovery.

2.2.2 Platform Independent I/O Access Routines

The platform independent I/O access routines are ioc$read_io and ioc$write_io. These provide a platform independent way to read and write I/O space without the overhead of CRAM allocation and initialization. These routines require that the I/O space that is to be accessed has been previously mapped by a call to ioc$map_io.

With the new mapping and access routines, we have the following basic model of I/O bus access:

- Map the device into the processor address space: Do the mapping yourself based on knowledge of a specific platform and bus OR use the new routine IOC$MAP_IO.
- Access the device: Do it yourself based on platform details, use CRAMS, or using the new platform independent access routines.

IOC$READ_IO and IOC$WRITE_IO are supported on PCI, EISA, Turbochannel, and Futurebus+. These routines are not supported on XMI.

2.3 Accessing Registers Directly

Registers that are mapped into the processors' virtual address space and accessed with load and store instructions are said to be accessed directly. This is similar to VAX-style I/O register access. On an AXP system, registers that are implemented on hardware directly connected to the processor-memory interconnect are usually accessed in this manner. Sparse space and swizzle space register access are examples of direct I/O device register access.

2.4 Accessing Registers Using CRAMS

Hardware I/O mailboxes exist only on DEC4000 Series and DEC7000/DEC10000 Series computers. The CRAM data structure and associated routines and IOC$READIO and IOC$WRITE_IO hide the underlying hardware mechanism (swizzle space, sparse space, or hardware I/O mailbox) from the programmer.

In addition to the CRAM data structure, OpenVMS AXP provides a set of system routines and corresponding macros that, on behalf of a device driver, allocate and initialize CRAMs. Table 2–1 lists these routines and macros. For more information about each system routine and macro, see *OpenVMS AXP Device Support: Reference*. Subsequent sections of this chapter describe driver mailbox operations in more detail.

Table 2–1 OpenVMS Macros and System Routines That Manage I/O Mailbox Operations

Routine	Macro	Description
IOC$ALLOCATE_ CRAM	DPTAB **idb_crams**, **ucb_crams** CRAM_ALLOC	Allocates and initializes a CRAM
IOC$CRAM_CMD	CRAM_CMD	Generates values for the command, mask, and remote I/O interconnect address (RBADR) fields of a CRAM
IOC$CRAM_IO	CRAM_IO	Issues the I/O space transaction defined by the CRAM.
IOC$DEALLOCATE_ CRAM	CRAM_DEALLOC	Deallocates a CRAM

2.5 Allocating CRAMs

A driver can use the following basic CRAM allocation strategies:

- Allocate a CRAM for every register the driver ever needs to access.
- Allocate a CRAM and reuse it.
- A driver can preallocate CRAMs at driver loading, or in a driver controller or unit initialization routine, linking them to a list connected to a UCB, IDB, or some driver-specific structure. This strategy is optimal for drivers that use CRAMs in performance-sensitive code.

- A driver can reuse and rebuild CRAMs as needed. Although fewer CRAMs suffice for the purposes of such a driver, this strategy is best suited for access to registers that are not in a performance sensitive code path. drivers that are less performance-sensitive.

Even though a driver can reuse CRAMs, a driver should not reuse a CRAM until it has checked the return status from IOC$CRAM_IO.

2.5.1 Preallocating CRAMs to a Device Unit or Device Controller

An OpenVMS AXP device driver can preallocate CRAMs and store them in a linked list associated with some data structure. It accomplishes this by repeatedly calling IOC$ALLOCATE_CRAM and inserting the address of the CRAM returned by this routine in the CRAM list. Or, CRAMS can be automatically preloaded by driver loading as described here.

Drivers often preallocate CRAMs to perform I/O operations on device unit registers or device controller registers. To facilitate the allocation of CRAMs for these purposes, the OpenVMS AXP driver loading procedure examines two fields in the DPT, DPT$W_IDB_CRAMS and DPT$W_UCB_CRAMS, for an indication of how many CRAMs the driver plans on using. Although the default value of both fields is zero, you can insert the number of CRAMs a driver requires to address device unit registers and device controller registers by specifying the **idb_crams** and **ucb_crams** arguments in the driver's DPTAB macro invocation. IDB CRAMs are available for use by a controller or unit initialization routine; UCB CRAMs are available for use by a unit initialization routine.

The driver loading procedure calls IOC$ALLOCATE_CRAM for each requested CRAM and inserts it in either of two singly linked lists: UCB$PS_CRAM as the header of a list of device unit CRAMs, and IDB$PS_CRAM as the header of a list of device controller CRAMs.

2.5.2 Calling IOC$ALLOCATE_CRAM to Obtain a CRAM

To allocate a single CRAM, a driver makes a standard call to IOC$ALLOCATE_CRAM, specifying a location to receive the address of the allocated CRAM and, optionally, the addresses of the IDB, UCB, or ADP.

IOC$ALLOCATE_CRAM allocates the CRAM and initializes it as follows:

CRAM$W_SIZE	Size of CRAM structure in bytes
CRAM$B_TYPE	Structure type (DYN$C_MISC)
CRAM$B_SUBTYPE	Structure type (DYN$C_CRAM)
CRAM$Q_RBADR	Address of remote I/O interconnect location (from IDB$Q_CSR)
CRAM$B_HOSE	Remote I/O interconnect number (from ADP$B_HOSE_NUM)
CRAM$L_IDB	IDB address
CRAM$L_UCB	UCB address

Normally, an OpenVMS AXP device driver can use the DPTAB macro to allocate CRAMs and associate them with a UCB or IDB; drivers that need to associate CRAMs with other structures may elect to allocate them from within a suitable fork thread.

IOC$ALLOCATE_CRAM cannot be called from above IPL$_SYNCH. Therefore, controller and unit initialization routines (which are called by the driver-loading procedure at IPL$_POWER) cannot allocate CRAMs. For CRAMS needed in or managed by controller or unit initialization routines, Digital recommends the DPTAB parameters as the means for CRAM allocation.

2.6 Constructing a Mailbox Command Within a CRAM

Once it has allocated CRAMs for its operations on device registers, an OpenVMS AXP device driver initializes each CRAM, so that it can use the CRAM in a transaction to a device interface register.

A driver initializes a CRAM by issuing a standard call to IOC$CRAM_CMD, specifying the **cmd_index**, **byte_offset**, and **adp_ptr**, and **cram_ptr iohandle** arguments. IOC$CRAM_CMD uses the input parameters supplied in the call to generate values for the command, mask, and I/O bus address fields of the CRAM that are specific to the bus that is the target of the mailbox operation.

Use the **cmd_index** argument to indicate the size and type of the register operation the mailbox describes. Although the $CRAMDEF macro (in SYS$LIBRARY:LIB.MLB) defines the command indices listed in Table 2–2, the actual commands supported under a given processor–I/O subsystem configuration vary from configuration to configuration. (Your specification of the **adp** argument allows IOC$CRAM_CMD to find the location of the command table that corresponds to a given I/O interconnect.) If you specify a command index that does not correspond to a supported command on the current system, IOC$CRAM_CMD returns SS$_BADPARAM status.

Table 2–2 Mailbox Command Indices Defined by $CRAMDEF

Command Index	Description
CRAMCMD$K_RDQUAD32	Quadword read in 32-bit space
CRAMCMD$K_RDLONG32	Longword read in 32-bit space
CRAMCMD$K_RDWORD32	Word read in 32-bit space
CRAMCMD$K_RDBYTE32	Byte read in 32-bit space
CRAMCMD$K_WTQUAD32	Quadword write in 32-bit space
CRAMCMD$K_WTLONG32	Longword write in 32-bit space
CRAMCMD$K_WTWORD32	Word write in 32-bit space
CRAMCMD$K_WTBYTE32	Byte write in 32-bit space
CRAMCMD$K_RDQUAD64	Quadword read in 64 bit space
CRAMCMD$K_RDLONG64	Longword read in 64 bit space
CRAMCMD$K_RDWORD64	Word read in 64 bit space
CRAMCMD$K_RDBYTE64	Byte read in 64 bit space
CRAMCMD$K_WTQUAD64	Quadword write in 64 bit space
CRAMCMD$K_WTLONG64	Longword write in 64 bit space
CRAMCMD$K_WTWORD64	Word write in 64 bit space
CRAMCMD$K_WTBYTE64	Byte write in 64 bit space

Use the **byte_offset** argument to specify the location of the device register that is the object of the mailbox command. Include the **cram** argument to identify the CRAM that contains the hardware I/O mailbox fields IOC$CRAM_CMD is to initialize.

Before using the hardware I/O mailbox in a write transaction to a device interface register, the driver must insert the data to be written to the register into CRAM$Q_WDATA.

2.6.1 Register Data Byte Lane Alignment

The CRAM routines supplied by OpenVMS AXP enforce a **longword oriented** view of I/O adapter register space, which means that adapter register space is viewed as if register bytes occupy a 32 bit data path, as follows:

```
              Adapter Register space

     31    24 23  16 15   8 7     0   offset

       byte 3  byte 2 byte 1  byte 0    0
       byte 7  byte 6 byte 5  byte 4    4

                   etc
```

Write example: To write a byte to register byte 2, specify IOC$CRAM_CMD parameters as follows:

```
command_index = cramcmd$k_wtbyte32

  byte_offset = 2
  adp_address = adp address
  cram_address = cram address
```

The data to be written must be positioned in bits 23:16 of the write data field (CRAM$Q_WDATA).

Read example: To read a byte from register byte 2, specify IOC$CRAM_CMD parameters as above except use cramcmd$k_rdbyte32 as the command_index.

The data from register byte 2 will be returned in bits 23:16 of the CRAM read data field (CRAM$Q_RDATA).

The programmer must perform the proper byte lane alignment of data for register writes. On register reads, the data is returned in its natural byte lane without any shifting. Note that this way of looking at adapter register space maps directly to the semantics of most I/O buses, but is distinctly diferent from VAX behavior.

2.7 Initiating a Mailbox Transaction

An OpenVMS AXP device driver initiates to a device register by issuing a standard call to IOC$CRAM_IO.

2.8 I/O Device Register Access Summary

This chapter explains the difference between direct register access and mailbox register access, and described the OpenVMS AXP routines and data structures that support register access. It should be noted again that the CRAM data structures and routines exist for all platforms and buses, regardless of whether or not the I/O subsystem hardware actually supports hardware mailboxes. The CRAM should be viewed simply as a data structure that describes an I/O register reference. The use of CRAM data structures and routines for I/O register accesses contributes to driver portability, as most platform and bus implementation differences can be hidden from the driver writer.

3 Suspending Driver Execution

An OpenVMS VAX device driver can explicitly or indirectly cause itself to be suspended by invoking a VAX MACRO macro or by calling one of the OpenVMS system routines listed in Table 3–1. An OpenVMS driver fork process typically is suspended to accomplish one of the following tasks:

- To wait to obtain a system resource, such as a controller channel
- To wait for a device interrupt or timeout
- To resume its execution at a lower interrupt priority level (IPL), that is, to fork

Table 3–1 OpenVMS VAX Macros and System Routines That Suspend Driver Execution

Routine	Macro	Description
IOC$REQPCHANH, IOC$REQPCHANL	REQPCHAN	Requests a controller's primary data channel
IOC$WFIKPCH, IOC$WFIRLCH	WFIKPCH, WFIRLCH	Suspends a driver fork thread and folds its context into a fork block in anticipation of a device interrupt or timeout
EXE$FORK, EXE$IOFORK	FORK, IOFORK	Creates a fork process
EXE$FORK_WAIT	FORK_WAIT	Inserts a fork block on the fork-and-wait queue

An OpenVMS VAX system routine accomplishes the suspension by removing the fork routine address from 4(SP) and placing it (with the current contents of R3 and R4) into the fork block. The system routine then returns to its caller's caller at the address provided at 8(SP). In compliance with the OpenVMS calling standard, the MACRO-32 compiler for OpenVMS AXP, like other AXP compilers, cannot allow such absolute control over the stack. A typical routine written in VAX MACRO, and compiled for execution on an OpenVMS AXP system, begins with compiler-generated register saves and ends with register restores. To ensure that saved registers and the state of the stack are restored, a routine must execute this return code. Explicit control of the stack and the caller's caller form of return are not possible on OpenVMS AXP systems.

Consequently, in creating an OpenVMS AXP device driver, you must inspect the occasions in which the driver uses the VAX MACRO macros and routines listed in Table 3–1 to determine to which of the following categories they belong:

- Simple fork process

  The driver and its fork thread share only the context currently preserved across the suspension by the OpenVMS VAX routine or macro; namely, the fork routine address and the contents of R3 and R4.

- Kernel process

The driver and its fork thread save and restore stack regions that might contain routine return addresses. Typically such a driver executes subroutine calls (by means of a JSB instruction), saves the return address in a data structure, and calls an OpenVMS suspension routine. Drivers based on the class/port structure generally must use the OpenVMS kernel process services.

The kernel process mechanism enables a system context thread of execution to run on its own private stack. While a kernel process is stalled, it can leave its execution state on the stack, such as nested stack frames and saved registers. This ability to save execution state across a stall is the primary motivation for kernel processes. It simplifies driver algorithms that are naturally expressed as nested subroutine calls and that would otherwise require complex state descriptions. See Section 3.2 for a discussion of the OpenVMS kernel process mechanism.

3.1 Using the Simple Fork Process Mechanism

An OpenVMS AXP driver uses the OpenVMS simple fork process mechanism when it and its fork thread share only the context currently preserved across the suspension by the OpenVMS VAX routine or macro; namely, the fork routine address and the contents of R3 and R4. The caller of the OpenVMS suspension routine and the fork routine must not share stack regions or store routine return addresses in data structures.

To employ the simple fork process mechanism, an OpenVMS AXP driver uses the macros listed in Table 3–2. New parameters have been added to the FORK, IOFORK, FORK_WAIT, WFIKPCH, and WFIRLCH macros to minimize the need to make explicit calls to the AXP system-specific suspension routines.

OpenVMS AXP supports JSB-based fork routines as well as standard call-based fork routines. The new ENVIRONMENT parameter specifies if the macro is being invoked from within a JSB or CALL interface routine. The default value of the environment parameter is JSB because this supports usage that is most similar to OpenVMS VAX use of these macros. The remainder of Section 3.1 focuses on the differences between the OpenVMS simple fork mechanism and the OpenVMS AXP simple fork mechanism for the JSB environment. See Section 7.4 for a discussion of the additional differences that apply when the simple fork mechanism is used in a CALL environment.

Table 3–2 Macros That Suspend OpenVMS AXP Driver Execution

OpenVMS VAX Macro	OpenVMS AXP Macro	Function
FORK	FORK *[routine] [,continue] [,environment=JSB]*	Calls EXE$PRIMITIVE_FORK or EXE_STD$PRIMITIVE_FORK to create a simple fork process on the current processor
FORK_WAIT	FORK_WAIT *[routine] [,continue] [,environment=JSB]*	Calls EXE$PRIMITIVE_FORK_WAIT or EXE_STD$PRIMITIVE_FORK_WAIT to insert a fork block on the system fork-and-wait queue

(continued on next page)

Table 3–2 (Cont.) Macros That Suspend OpenVMS AXP Driver Execution

OpenVMS VAX Macro	OpenVMS AXP Macro	Function
IOFORK	IOFORK *[routine] [,continue] [,environment=JSB]*	Disables timeouts from the associated device and calls EXE$PRIMITIVE_FORK or EXE_STD$PRIMITIVE_FORK to create a fork process
REQPCHAN *[pri=LOW]*	REQCHAN *[pri=LOW] [,environment=JSB]*	Calls IOC_STD$PRIMITIVE_REQCHANH or IOC_STD$PRIMITIVE_REQCHANL to obtain a controller's data channel
WFIKPCH *excpt [,time=65536]* WFIRLCH *excpt [,time=65536]*	WFIKPCH *excpt [,time=65536] [,newipl][,environment=JSB]* WFIRLCH *excpt [,time=65536] [,newipl][,environment=JSB]*	Calls IOC_STD$PRIMITIVE_WFIKPCH or IOC_STD$PRIMITIVE_WFIRLCH to suspend a driver fork thread and folds its context into a fork block in anticipation of a device interrupt or timeout

Table 3–3 lists the system routines that an OpenVMS AXP driver uses to suspend execution.

Table 3–3 System Routines That Suspend OpenVMS AXP Driver Execution

OpenVMS VAX Routine	OpenVMS AXP Routine	Function
EXE$FORK	EXE$PRIMITIVE_FORK and EXE_STD$PRIMITIVE_FORK	Creates a simple fork process on the current processor
EXE$FORK_WAIT	EXE$PRIMITIVE_FORK_WAIT and EXE_STD$PRIMITIVE_FORK_WAIT	Inserts a fork block on the system fork-and-wait queue
EXE$IOFORK	EXE$PRIMITIVE_FORK and EXE_STD$PRIMITIVE_FORK	Creates a simple fork process on the local processor
IOC$REQPCHANH IOC$REQPCHANL	IOC_STD$PRIMITIVE_REQCHANH IOC_STD$PRIMITIVE_REQCHANL	Obtains a controller's data channel
IOC$WFIKPCH IOC$WFIRLCH	IOC_STD$PRIMITIVE_WFIKPCH IOC_STD$PRIMITIVE_WFIRLCH	Suspends a driver fork thread and folds its context into a fork block in anticipation of a device interrupt or timeout

3.1.1 EXE_STD$PRIMITIVE_FORK, EXE_STD$PRIMITIVE_FORK_WAIT, and Associated Macros

EXE$PRIMITIVE_FORK and EXE_STD$PRIMITIVE_FORK are the OpenVMS AXP counterpart to the OpenVMS VAX system routines EXE$FORK and EXE$IOFORK. EXE_STD$PRIMITIVE_FORK_WAIT is the OpenVMS AXP counterpart to the OpenVMS VAX EXE$FORK_WAIT routine.

Use of the simple fork process mechanism in an OpenVMS AXP device driver requires that you alter each instance of EXE$FORK, EXE$IOFORK, or EXE$FORK_WAIT in driver code by:

- Replacing each explicit JSB to EXE$FORK with either an invocation of the FORK macro or a JSB to EXE$PRIMITIVE_FORK. (Note that EXE$PRIMITIVE_FORK requires different inputs than EXE$FORK.)
- Replacing each explicit JSB to EXE$IOFORK with either an invocation of the IOFORK macro or with an instruction that clears UCB$V_TIM in UCB$L_STS followed by a JSB to EXE$PRIMITIVE_FORK.
- Replacing each explicit JSB to EXE$FORK_WAIT with either an invocation of the FORK_WAIT macro or a JSB to EXE$PRIMITIVE_FORK_WAIT. (Note that EXE$PRIMITIVE_FORK_WAIT requires different inputs than EXE$FORK_WAIT.)

For information about the calling conventions for EXE$PRIMITIVE_FORK and EXE$PRIMITIVE_FORK_WAIT see *OpenVMS AXP Device Support: Reference*.

The OpenVMS AXP versions of the FORK, IOFORK, and FORK_WAIT macros have been designed to conceal many of the differences between the behavior of the OpenVMS VAX and the OpenVMS AXP routines for most device drivers. The following sections provide some examples of how an OpenVMS AXP device driver may use these macros. *OpenVMS AXP Device Support: Reference* provides more information about the use and operation of the FORK and IOFORK macros.

3.1.1.1 Common Usage of the FORK and IOFORK Macros

Drivers most commonly use the FORK and IOFORK macros in situations where execution is to be resumed at the caller's caller when the fork block is queued, and where the fork routine's entry point immediately follows the invocation of the macro. A FORK or IOFORK macro invocation of this type needs no change to work properly in an OpenVMS AXP device driver.

Consider the following OpenVMS driver source:

```
r:      code_a
        iofork
        code_b
        rsb
```

It has the following expansion on an OpenVMS VAX system:[1]

```
r:      code_a
        JSB     G^EXE$IOFORK
        code_b
        rsb
```

The effect is that the first instruction of *code_b* is queued as a fork routine and that EXE$IOFORK returns directly to the caller of routine *r*.

It has the following expansion on an OpenVMS AXP system:

[1] Original source is shown in lowercase and the results of macro expansion are shown in uppercase.

```
r:      code_a
        BICL    #UCB$M_TIM,UCB$L_STS(R5)
        MOVAB   F,FKB$L_FPC(R5)
        JSB     G^EXE$PRIMITIVE_FORK
        RSB
F:      .JSB_ENTRY INPUT=<R3,R4,R5>,SCRATCH=<R0,R1,R2,R3,R4>
        code_b
        rsb
```

The effect is the same as the OpenVMS VAX expansion. The fork routine is defined to begin with the first instruction of *code_b*; *F* is the generated label for the fork routine. Control is returned to the caller of *r* by means of the explicit RSB that is generated after the JSB to EXE$PRIMITIVE_FORK.

Note

On OpenVMS AXP systems, any branch between *code_a* and *code_b* must obey the restrictions of cross-routine branches, as described in Chapter 6. Meeting these restrictions may require source changes. For more information, see *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code*.

3.1.1.2 Forks with Nonstandard Returns and Nonstandard Fork Routine Addresses

Some direct calls to EXE$FORK or EXE$IOFORK require either a nonstandard continue label, nonstandard fork routine address, or both.

The OpenVMS AXP versions of the FORK and IOFORK macros provide two optional arguments that allow drivers to specify these items and avoid a direct call to EXE$PRIMITIVE_FORK:

- The **continue** argument specifies the label where execution continues after the fork block has been inserted on the fork queue. If you omit this argument, control returns to the caller of the routine that invoked the FORK or IOFORK macro.
- The **routine** argument specifies the name of the routine to be executed in fork context. If you omit this argument, the macro assumes that the fork routine immediately follows the FORK or IOFORK macro invocation.

Example of Nonstandard Return from Fork Operation

In the following example, the OpenVMS VAX driver that is calling EXE$IOFORK wants to queue the fork thread and return control back to itself (that is, to label *l* in routine *r*) and not the caller's caller:

```
r:      code_a1
l:      code_a2
        pushab  l
        jsb     g^exe$iofork
        code_b
        rsb
```

In an OpenVMS AXP device driver, this code would be rendered as:

```
r:      code_a1
l:      code_a2
        iofork    continue=l
        code_b
        rsb
```

The expansion of this IOFORK macro invocation on an OpenVMS AXP system would be as follows:

```
r:      code_a1
l:      code_a2
        BICL    #UCB$M_TIM,UCB$L_STS(R5)
        MOVAB   F,FKB$L_FPC(R5)
        JSB     G^EXE$PRIMITIVE_FORK
        BRW     l
F:      .JSB_ENTRY INPUT=<R3,R4,R5>,SCRATCH=<R0,R1,R2,R3,R4>
        code_b
        rsb
```

Example of Nonstandard Fork Routine Address

The following code excerpt from an OpenVMS VAX device driver illustrates the case where the fork routine (that is, *fr*) is not located in the source immediately after the call to EXE$IOFORK:

```
r:      code_a1
        pushab  fr
        jmp     g^exe$iofork
       .
       .
       .
fr:     code_b
        rsb
```

In an OpenVMS AXP device driver, this code would be as follows:

```
r:      code_a1
        iofork    routine=fr
        .
        .
        .
fr:     fork_routine
        code_b
        rsb
```

Note that, because the IOFORK macro cannot automatically add the entry point directive at the start of a fork routine that may be located anywhere, you must manually add the new FORK_ROUTINE macro to the source.

The expansion of the FORK_ROUTINE macro would be as follows:

```
.JSB_ENTRY  INPUT=<R3,R4,R5>,SCRATCH=<R0,R1,R2,R3,R4>
```

The expansion of the IOFORK macro invocation on an OpenVMS AXP system would be as follows:

```
r:      code_a1
        BICL    #UCB$M_TIM,UCB$L_STS(R5)
        MOVAB   fr,FKB$L_FPC(R5)
        JSB     G^EXE$PRIMITIVE_FORK
        RSB
       .
       .
       .
fr:     fork_routine
        code_b
        rsb
```

3.1.2 IOC_STD$PRIMITIVE_REQCHANH, IOC_STD$PRIMITIVE_REQCHANL, and the REQCHAN Macro

IOC_STD$PRIMITIVE_REQCHANH and IOC_STD$PRIMITIVE_REQCHANL are the OpenVMS AXP counterparts to the OpenVMS VAX system routines IOC$REQPCHANH and IOC$REQPCHANL.

Use of the simple fork process mechanism in an OpenVMS AXP device driver requires that you replace each explicit JSB to IOC$REQPCHANH or IOC$REQPCHANL with an invocation of the REQPCHAN[2] or REQCHAN macro.

Note

IOC$REQSCHANH and IOC$REQSCHANL are not supported in OpenVMS AXP systems because the concept of primary and secondary controller channels is not meaningful in the I/O subsystem.

For more information about the calling conventions for IOC_STD$PRIMITIVE_REQCHANH and IOC_STD$PRIMITIVE_REQCHANL, see *OpenVMS AXP Device Support: Reference*.

The OpenVMS AXP versions of the REQPCHAN and REQCHAN macros have been designed to conceal many of the differences between the behavior of the OpenVMS VAX and the OpenVMS AXP routines for most device drivers.

Consider the following OpenVMS driver source:

```
r:      code_a
        reqpchan
        code_b
        rsb
```

This code example expands in the following way on an OpenVMS AXP system:

```
r:      code_a
        MOVAB   F,FKB$L_FPC(R5)
        SUBL    #4,SP
        PUSHAB  (SP)
        PUSHL   R5
        PUSHL   R3
        CALLS   #3,G^IOC_STD$PRIMITIVE_REQCHANL
        POPL    R4
        BLBS    R0,L
        RSB
F:      .JSB_ENTRY INPUT=<R3,R4,R5>,SCRATCH=<R0,R1,R2,R3,R4>
L:      code_b
        rsb
```

The effect of the resulting code is the same as the OpenVMS VAX expansion. The fork routine is defined to begin with the first instruction of *code_b*; *F* is the generated label for the fork routine. If the channel is immediately assigned to the driver, execution continues at the generated label *L* at the first instruction of *code_b*. Otherwise, control is returned to the caller of *r* by means of the explicit RSB that is generated after the CALL to IOC_STD$PRIMITIVE_REQCHANL. When the channel is eventually assigned to the driver, IOC_STD$RELCHAN calls fork routine *F*.

[2] The REQPCHAN macro is provided for compatibility with OpenVMS VAX; use of the REQCHAN macro is preferred with OpenVMS AXP.

Note

Any branches between *code_a* and *code_b* must obey the restrictions of crossroutine branches, as described in Chapter 6. Meeting these restrictions may require source changes. Also, the macro contains a branch between *code_a* and *code_b*.

See *OpenVMS AXP Device Support: Reference* for additional information on the use and operation of the REQCHAN macro.

3.1.3 IOC_STD$PRIMITIVE_WFIKPCH, IOC_STD$$PRIMITIVE_WFIRLCH, and Associated Macros

IOC_STD$PRIMITIVE_WFIKPCH and IOC_STD$PRIMITIVE_WFIRLCH are the OpenVMS AXP counterparts to the OpenVMS VAX system routines IOC$WFIKPCH and IOC$WFIRLCH. For more information about the calling conventions for IOC_STD$PRIMITIVE_WFIKPCH and IOC_STD$PRIMITIVE_WFIRLCH, see *OpenVMS AXP Device Support: Reference*.

The OpenVMS AXP versions of the WFIKPCH and WFIRLCH macros have been designed to conceal many of the differences between the behavior of the OpenVMS VAX and the OpenVMS AXP routines for most device drivers.

- The **excpt** argument specifies the label of the timeout handling code within the driver. On an OpenVMS VAX system, EXE$TIMEOUT calls a driver's timeout handling routine directly by means of a VAX MACRO JSB instruction. On an OpenVMS AXP system, EXE$TIMEOUT calls the driver time out routine (at UCB$PS_TOUTROUT) with UCB$V_TIMOUT set. If the TOUTROUT parameter is blank, then the WFIKPCH and WFIRLCH macros use the fork routine for the timeout routine as well.

  These macros automatically insert an instruction at the beginning of the fork routine that tests UCB$V_TIMOUT in UCB$L_STS and branches to the label of the timeout code if it is set.

- The WFIKPCH and WFIRLCH macros automatically place the procedure value of the fork routine (at the instruction following the macro invocation) in UCB$L_FPC.

- The **time** argument expresses the timeout interval in seconds as on OpenVMS VAX systems.

- The **newipl** argument specifies the IPL to which the wait-for-interrupt routine should lower before the wait-for-interrupt macro returns to its caller. Typically this is the fork IPL associated with device processing that was pushed on the stack by a prior invocation of the DEVICELOCK macro. If you omit this argument, the macro considers the value on the top of the stack as the return IPL. This default allows an OpenVMS AXP driver to use the macro in the same way as an OpenVMS VAX driver does.

- The **toutrout** argument specifies a timeout routine address.

Example of WFIKPCH with Default newipl Argument

The following code example illustrates how a standard invocation of the WFIKPCH macro in an existing OpenVMS driver needs no change to work properly in an OpenVMS AXP device driver.

```
   r: code_a1
devicelock      -
        lockaddr=ucb$l_dlck(r5),-
        savipl=-(sp)
code_a2
wfikpch tmo_label,#tmo
code_b
rsb
```

On an OpenVMS AXP system, this code example expands as follows:

```
r: code_a1
     devicelock      -
             lockaddr=ucb$l_dlck(r5),-
             savipl=-(sp)
     code_a2
     MOVL    #tmo,R1
     MOVL    (SP)+,R2
     MOVAB   F,UCB$L_FPC(R5)
     MOVAB   F,UCB$PS_TOUTROUT(R5)
     PUSHL   R2
     PUSHL   R1
     PUSHL   R5
     PUSHL   R4
     PUSHL   R3
     CALLS   #5,IOC_STD$PRIMITIVE_WFIKPCH
     RSB
F:   .JSB_ENTRY INPUT=<R3,R4,R5>,SCRATCH=<R0,R1,R2,R3,R4>
     BITL    #UCB$M_TIMOUT,UCB$L_STS(R5)
     BNEQ    tmo_label
     code_b
     rsb
```

Example of WFIKPCH Specifying newipl Argument

The following code example has the same effect as the first. It accomplishes this by saving the original IPL directly into R2 using the DEVICELOCK macro, and later specifying R2 as the **newipl** argument to WFIKPCH.

```
r:      code_a1
        devicelock      -
                lockaddr=ucb$l_dlck(r5),-
                savipl=r2
        code_a2
        wfikpch tmo_label,#tmo,newipl=r2
        code_b
        rsb
```

On an OpenVMS AXP system, this code has the following expansion:

```
r:      code_a1
        devicelock      -
                lockaddr=ucb$l_dlck(r5),-
                savipl=r2
        code_a2
        MOVL    #tmo,R1
        MOVAB   F,UCB$L_FPC(R5)
        MOVAB   F,UCB$PS_TOUTROUT(R5)
        PUSHL   R2
        PUSHL   R1
        PUSHL   R5
        PUSHL   R4
        PUSHL   R3
        CALLS   #5,IOC_STD$PRIMITIVE_WFIKPCH
        RSB
F:      .JSB_ENTRY INPUT=<R3,R4,R5>,SCRATCH=<R0,R1,R2,R3,R4>
        BITL    #UCB$M_TIMOUT,UCB$L_STS(R5)
        BNEQ    tmo_label
        code_b
        rsb
```

See *OpenVMS AXP Device Support: Reference* for further details on the use and operation of the WFIKPCH and WFIRLCH macros.

3.2 Using the OpenVMS Kernel Process Services

The OpenVMS kernel process services enable a system context thread of execution to run on its own private stack. This thread of execution is known as a **kernel process**. Prior to suspending itself (to fork or to wait for an interrupt or controller channel), a kernel process stores its execution state (such as register contents) on its private stack (which may include the nested stack frames of previous procedure calls within the kernel process). When it is resumed, a kernel process has access to the data that has previously been stored on its private stack.

The ability to save some execution state on a stack across a stall is the primary motivation for kernel processes. It simplifies driver algorithms that are naturally expressed as nested subroutine calls and that would otherwise require complex state descriptions. Also, this ability is a prerequisite to supporting device drivers written in a high level language.

Two data structures describe a kernel process. Typically, an OpenVMS AXP device driver calls a system routine to create these data structures when it initiates a kernel process and calls another routine to delete them when the kernel process has completed.

- A **kernel process block** (KPB) that describes the context and state of a kernel process
- A stack that records the current state of execution of the kernel process

The KPB consists of the following areas:

- Base area

  The base area includes the standard OpenVMS data structure header fields, describes the kernel process private stack, contains masks that describe the KPB itself and its register saveset, stores the context of a suspended KPB, and provides pointers to the other KPB areas. The KPB base area ends with offset KPB$IS_PRM_LENGTH.
- Scheduling area

The scheduling area contains the procedure values of the routines that execute to suspend a kernel process and to resume its execution. The scheduling area can contain either a fork block or a timer queue entry. The scheduling area ends with offset KPB$Q_FR4.

- OpenVMS special parameters area

  The OpenVMS special parameters area stores information required by OpenVMS device drivers, such as pointers to I/O database structures, data facilitating the selection and operation of driver macros, and driver-specific data. The OpenVMS special parameters area ends with offset KPB$PS_DLCK.

- Spin lock area

  The spin lock area is unused at present and reserved to Digital. It ends with offset KPB$PS_SPL_RESTRT_RTN.

- Debugging area

  The debugging area stores information used in the debugging of a kernel process. The KPB debugging area follows either the scheduling or spin lock area.

- Parameter area

  The parameter area is a variably-sized area that is specified by the kernel process creator in the call to EXE$KP_ALLOCATE_KPB. The kernel process creator and the kernel process use this area to exchange data.

The KPB can be used in one of two general types: the OpenVMS executive software type (VEST) and the fully general type (FGT). OpenVMS software always uses the VEST form of the KPB.

In a VEST KPB, the base, scheduling, OpenVMS special parameters, and spin lock areas have a fixed position relative to the starting address of the KPB. This allows you to access all fields in these areas as offsets from a single register that points to the KPB's starting address.

Entry into and exit from a kernel process always involves a stack switch. During execution as a kernel process, a system context thread of execution, such as a process fork, calls a set of OpenVMS provided routines that preserve register context and switch stacks:

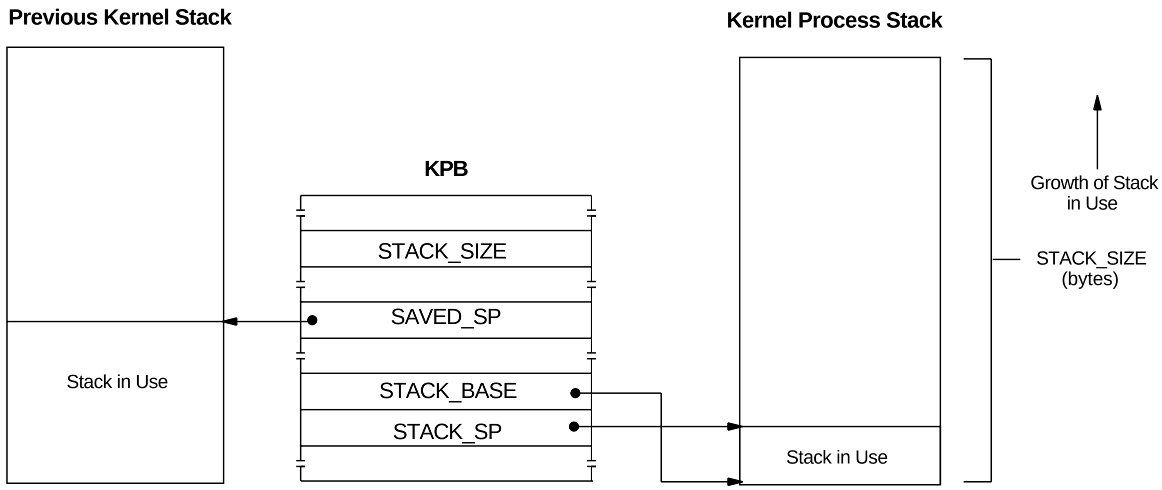
- At initiation, a switch from the current kernel stack to that of the kernel process
- At a stall, a switch from the kernel process private stack to the one current when the kernel process was entered
- At restart, a switch from the current kernel stack to that of the kernel process
- At termination, a switch from the kernel process private stack to the one current when the kernel process was most recently entered

As shown in Figure 3–1 KPBIS_STACK_SIZE, KPBPS_STACK_BASE, and KPB$PS_STACK_SP describe the kernel process stack. KPB$PS_SAVED_SP contains the stack pointer on the stack current when the kernel process was initiated or restarted. That pointer is restored when the kernel process stalls or terminates.

A kernel process private stack occupies one or more pages of system space allocated for that purpose when the kernel process is created. The stack has a no-access guard page at each end so that stack underflow and overflow can be detected immediately.

Figure 3–1 shows the stack and the fields in the KPB related to it.

Figure 3–1 Kernel Process Private Stack



3.2.1 Kernel Process Routines

The routines (and associated macros) listed in Table 3–4 create a kernel process and its associated structures, and maintain the kernel process environment. A driver that specifies in its DDT EXE_STD$KP_STARTIO as its start-I/O routine creates a kernel process in which its own start-I/O routine runs. (Alternatively, the driver can make successive calls to EXE$KP_ALLOCATE_KPB and EXE$KP_START to accomplish the same result.)

Once executing as a kernel process, in order to stall, the thread must call a routine that can switch stacks and then save the thread's state in such a way that it can restart when the stall ends. The kernel process can call any of the supplied scheduling stall routines (EXE$KP_STALL_GENERAL, EXE$KP_FORK, EXEKP_FORK_WAIT, IOCKP_REQCHAN, IOC$KP_WFIKPCH, and IOC$KP_WFIRLCH), or invoke any of the corresponding macros, to safely suspend its execution. When the condition implied in the stall request is met (for instance, a device interrupt or the grant of a controller channel), OpenVMS calls EXE$KP_RESTART to resume execution of the kernel process.

If a driver kernel process was created by EXE_STD$KP_STARTIO, it requests its own termination as part of request completion, by invoking the KP_REQCOM macro.

Table 3–4 System Routines and Macros That Create and Manage Kernel Processes

System Routine	Driver Macro	Function
EXE_STD$KP_STARTIO	DDTAB (**start**=EXE_STD$KP_STARTIO, **kp_startio**=driver-start-IO-routine)	Allocates and sets up a KPB and a kernel process private stack, and starts up the execution of a kernel process used by a device driver
EXE$KP_ALLOCATE_KPB	KP_ALLOCATE_KPB DDTAB (**start**=EXE_STD$KP_STARTIO, **kp_startio**=driver-start-IO-routine)	Allocates a KPB and its kernel process private stack
EXE$KP_START	KP_START DDTAB (**start**=EXE_STD$KP_STARTIO, **kp_startio**=driver-start-IO-routine)	Starts the execution of a kernel process
EXE$KP_STALL_GENERAL	KP_STALL_GENERAL KP_STALL_FORK KP_STALL_FORK_WAIT KP_STALL_IOFORK KP_STALL_REQCHAN KP_STALL_WFIKPCH KP_STALL_WFIRLCH	Stalls the execution of a kernel process
EXE$KP_FORK	KP_STALL_FORK KP_STALL_IOFORK	Stalls a kernel process in such a manner that it can be resumed by the OpenVMS fork dispatcher
EXE$KP_FORK_WAIT	KP_STALL_FORK_WAIT	Stalls a kernel process in such a manner that it can be resumed by the software timer interrupt service routine's examination of the fork-and-wait queue
IOC$KP_REQCHAN	KP_STALL_REQCHAN	Stalls a kernel process in such a manner that it can be resumed by the granting of a device controller channel
IOC$KP_WFIKPCH IOC$KP_WFIRLCH	KP_STALL_WFIKPCH KP_STALL_WFIRLCH	Stalls a kernel process in such a manner that it can be resumed by device interrupt processing
EXE$KP_RESTART	KP_RESTART	Resumes the execution of a kernel process
EXE$KP_END	KP_END	Terminates the execution of a kernel process
EXE$KP_DEALLOCATE_KPB	KP_DEALLOCATE_KPB	Deallocates a KPB and its kernel process private stack

Because the kernel process routines (and macros) operate on subroutine call semantics, all return status in R0. For the routines (and macros) that manipulate kernel process structures, such as EXE$KP_ALLOCATE_KPB and EXE$KP_START, a driver should inspect the status value and take appropriate action.

The sections that follow describe the operations required to set up and use a driver kernel process. For further information on a specific kernel process macro or routine, see *OpenVMS AXP Device Support: Reference*.

3.2.2 Creating a Driver Kernel Process

A driver typically creates a kernel process by specifying EXE_STD$KP_STARTIO in the **start** argument to the DDTAB macro. EXE_STD$KP_STARTIO allocates and initializes a VEST KPB and allocates a kernel process private stack, and then places the driver kernel process into execution, at the address indicated by the **kp_startio** argument to the DDTAB macro.

EXE_STD$KP_STARTIO customizes the kernel process environment specifically for driver kernel processes, facilitating the conversion of OpenVMS VAX drivers that use the simple fork process mechanism to OpenVMS AXP drivers. To this end, EXE_STD$KP_STARTIO performs the following tasks:

- Specifies to EXE$KP_ALLOCATE_KPB the size of the kernel process private stack in bytes. EXE_STD$KP_STARTIO supplies the minimum value of DDT$IS_STACK_BCNT or KPB$K_MIN_IO_STACK (currently 8KB). A driver contributes a value to DDT$IS_STACK_BCNT by specifying the **kp_stack_size** argument to the DDTAB macro.
- Specifies IRP$PS_KPB to EXE$KP_ALLOCATE_KPB as the target location of the KPB address.
- Specifies to EXE$KP_ALLOCATE_KPB a VEST-type KPB with scheduling and spin lock sections and indicates that the KPB should be deleted when the kernel process is terminated.
- Issues a standard call to EXE$KP_ALLOCATE_KPB.
- Inserts the address of the IRP in KPB$PS_IRP and the address of the UCB in KPB$PS_UCB.
- Specifies to EXE$KP_START a mask indicating which registers must be preserved across context switches between the private kernel process private stack and the kernel stack. This mask allows any registers that the kernel process uses, other than those calling standard defines as “scratch” to be saved across its suspension and resumption.

  This mask is the logical-OR of the value of DDT$IS_REG_MASK and the value of KPREG$K_MIN_IO_REG_MASK (which specifies R2 through R5, R12 through R15, and R26, R27, and R29). A driver contributes a value to DDT$IS_REG_MASK by specifying the **kp_reg_mask** argument to the DDTAB macro. EXE_STD$KP_STARTIO excludes any registers that are illegal in a kernel process register save mask: R0, R1, R16 through R25, R27, R28, R30, and R31 (KPREG$K_ERR_REG_MASK).
- Specifies to EXE$KP_START the value of DDT$PS_KP_STARTIO as the procedure value of the routine to be placed into execution in the driver kernel process. A driver contributes a value to DDT$PS_KP_STARTIO by specifying the **kp_startio** argument to the DDTAB macro.

For drivers ported from OpenVMS VAX, the following invocation of the DDTAB macro is sufficient to create a kernel process for most drivers and start execution of the driver’s start-I/O routine as a kernel process thread:

```
DDTAB   -
START=EXE_STD$KP_STARTIO,-
KP_STARTIO=xx_STARTIO,-
.
.
.
```

The driver's start I/O routine, *xx*_STARTIO in the preceding example, gains control as a result of the call from EXE$KP_START and receives one parameter, the address of the KPB. It obtains the addresses of the UCB and IRP from KPB$PS_UCB and KPB$PS_IRP, respectively:

```
xx_STARTIO:
        .CALL_ENTRY <R2,R3,R4,R5>
        MOVL    4(AP),R0                ; Get KPB address
        MOVL    KPB$PS_UCB(R0),R5       ; Get UCB address
        MOVL    KPB$PS_IRP(R0),R3       ; Get IRP address
```

Note that the preceding code example essentially discards the KPB address, by placing it in a scratch register, R0. EXE_STD$KP_STARTIO stores the KPB address in IRP$PS_KPB so that the KPB address can always be found there at anytime at any depth of subroutine call.

Note

The VEST KPB created by EXE$KP_ALLOCATE_KPB in response to the call from EXE_STD$KP_STARTIO may not be sufficient for a driver kernel process that must exchange a lot of data with its creator. VEST KPBs do not include the debugging or parameter areas. If a driver requires either of these areas in a VEST KPB, it should not specify EXE_STD$KP_STARTIO in the **start** argument of the DDTAB macro. Rather it must make explicit calls to EXE$KP_ALLOCATE_KPB and EXE$KP_START, as well as initialize the kernel process environment in a manner similar to that used by EXE_STD$KP_STARTIO.

See Section 3.2.5 for additional information on using the KPB parameter area.

3.2.3 Suspending a Kernel Process

Once a kernel process thread has been initiated, all functions that cause suspension of that thread of driver execution must use kernel process stalling semantics. For existing OpenVMS device drivers, written in VAX MACRO, that employ simple fork process semantics, this generally means adding the phrase "KP_STALL_" to the beginning of a standard driver stall macro (for instance, WFIKPCH becomes KP_STALL_WFIKPCH).

Table 3–5 contrasts the simple fork process and the kernel process suspension macros:

Table 3–5 Comparison of Simple Fork Process and Kernel Process Suspension Macros

Simple Fork Process Suspension Macro	Kernel Process Suspension Macro	When called
FORK	KP_STALL_FORK	When creating a fork thread
FORK_WAIT	KP_STALL_FORK_WAIT	When creating a short fork wait thread
IOFORK	KP_STALL_IOFORK	When creating a I/O fork thread

(continued on next page)

Table 3–5 (Cont.) Comparison of Simple Fork Process and Kernel Process Suspension Macros

Simple Fork Process Suspension Macro	Kernel Process Suspension Macro	When called
REQCHAN[1]	KP_STALL_REQCHAN	When requesting an I/O device channel
WFIKPCH	KP_STALL_WFIKPCH	When waiting for an interrupt or timeout
WFIRLCH	KP_STALL_WFIRLCH	When waiting for an interrupt or timeout
REQCOM[2]	KP_REQCOM	When completing an I/O request

[1]The KP_STALL_ macros provide no replacement for the REQPCHAN macro. When a driver uses kernel processes, REQPCHAN should be replaced with KP_STALL_REQCHAN.

[2]Replacing REQCOM with KP_REQCOM has no bearing on how a driver thread is stalled. It does provide for correct termination and cleanup of a driver kernel process thread upon completion of an I/O request. See Section 3.2.4.

The kernel process suspension macros all require as input the address of a KPB. For macros that replace traditional suspension macros in existing OpenVMS drivers, the R0 status is typically SS$_NORMAL, and thus not very interesting. However, newly written drivers should be coded to check return status values.

For further information on a specific kernel process suspension macro, see *OpenVMS AXP Device Support: Reference*.

3.2.4 Terminating a Kernel Process Thread

A driver kernel process initiated by EXE_STD$KP_STARTIO (in which the start-I/O routine is the top-level thread) is terminated properly by the KP_REQCOM macro (which includes a VAX MACRO RET instruction).

To ensure that the terminated KPB is released for future reuse, the flag KPB$V_DEALLOC_AT_END must be set in the KPB$IS_FLAGS field. If you are allocating a KPB via some mechanism other than EXE_STD$KP_STARTIO, you should ensure that this flag is set. EXE_STD$KP_STARTIO sets KPB$V_DEALLOC_AT_END.

3.2.5 Exchanging Data Between a Kernel Process and Its Creator

In the unlikely event that a driver kernel process requires more data than it can obtain from the KPB address (its sole input parameter), its creator can establish a parameter area in the KPB.

A driver creates a KPB with a parameter area by specifying the **param** argument to a KP_ALLOCATE_KPB macro invocation (or the **param_size** parameter to a call to EXE$KP_ALLOCATE_KPB).

The following example shows a simple exchange of data residing in the KPB parameter area between a kernel process and its creator:

```
KP_ALLOCATE_KPB   kpb=R2, param=#32        ;32-byte parameter area
MOVL    KPB$PS_PRM_PTR(R2),R1              ;Obtain pointer to parameter area
MOVL    R3,(R1)                            ;Save R3
MOVL    R4,4(R1)                           ;Save R4
KP_SWITCH_TO_KP_STACK                      ;Switch to KP stack
MOVL    KPB$PS_PRM_PTR(R6),R1              ;Obtain pointer to parameter area
MOVL    (R1),R3                            ;Obtain saved R3
MOVL    4(R1),R4                           ;Obtain saved R4
```

3.2.6 Synchronizing the Actions of a Kernel Process and Its Initiator

Neither the initiator of the kernel process (that is, the caller of EXE$KP_START or EXE$KP_RESTART) nor the kernel process itself can assume that there is any relationship between them unless they mutually establish one. The initiator and the kernel process must establish explicit synchronization between themselves for operations that require it.

The kernel process cannot assume that its initiator is not running in parallel. Neither can it depend on inheriting the synchronization capabilities of its caller (for instance, its spin locks and IPL). The initiator of the kernel process thread cannot assume that the kernel process has already executed when EXE$KP_START returns control.

3.2.7 Example of Driver Kernel Process

Example 3–2 shows an OpenVMS VAX simple driver start I/O routine of Example 3–1, modified to use the OpenVMS kernel process services.

Example 3–1 Simple Start I/O Routine

```
STARTIO:
          .
          .
; Initiate device activity by informing controller
; of required action
        .
        .
        .
          WFIKPCH   DEVTMO,#6     ;Wait for interrupt or timeout
          .                       ;Execution resumes here upon
          .                       ; interrupt
          .
          IOFORK                  ;Request to defer further
          .                       ; processing to a lower IPL
          .
          .
          REQCOM                  ;Initiate I/O request completion
                                  ; processing
```

To use the kernel process mechanism, a VAX MACRO device driver must adopt the following conventions. The numbers in the following list represent the contents of Example 3–2.

1. The DDTAB macro invocation must identify EXE_STD$KP_STARTIO as the **start** argument and the start-I/O routine within the driver as the **kp_startio** argument.
2. The start-I/O routine within the driver must be a standard-conforming procedure. Here, the start-I/O routine specifies the .CALL_ENTRY MACRO compiler directive with a typical driver register preserve mask (R2 through R5).
3. The start I/O procedure must retrieve the addresses of the IRP and UCB from the kernel process block (KPB) associated with the kernel process.

Example 3–2 Simple Start I/O Routine That Uses the Kernel Process Mechanism

```
         .
         .
         .
         DDTAB -
              START=EXE_STD$KP_STARTIO,- 1
              KP_STARTIO=STARTIO,-       ;Miscellaneous other required
                                         ; changes ignored
           .
           .
           .
STARTIO:   .CALL_ENTRY <R2,R3,R4,R5> 2
           MOVL   4(AP),R0               ;Get KPB address
           MOVL   KPB$PS_UCB(R0),R5      ;Get UCB address 3
           MOVL   KPB$PS_IRP(R0),R3      ;Get IRP address
           .
           .
           .
           KP_STALL_WFIKPCH DEVTMO,#6    ;Wait for interrupt 4
           .                             ; or timeout
           .
           .
           KP_STALL_IOFORK               ;Wait until IPL drops
           .                             ; to fork IPL
           .
           .
           KP_REQCOM                     ;Complete request
```

4 The start I/O procedure must use the KP_STALL_*xxx* or KP_*xxx* macros instead of the equivalent OpenVMS VAX macros.

The following is a brief description of the control flow of an I/O operation through the start-I/O routine shown in Example 3–2. Although the details of interaction between the start-I/O routine and the OpenVMS operating system are different from that which transpires between a driver simple fork process and the OpenVMS operating system, the overall structure of a driver that uses the kernel process mechanism is much the same as one that uses the simple fork process mechanism.

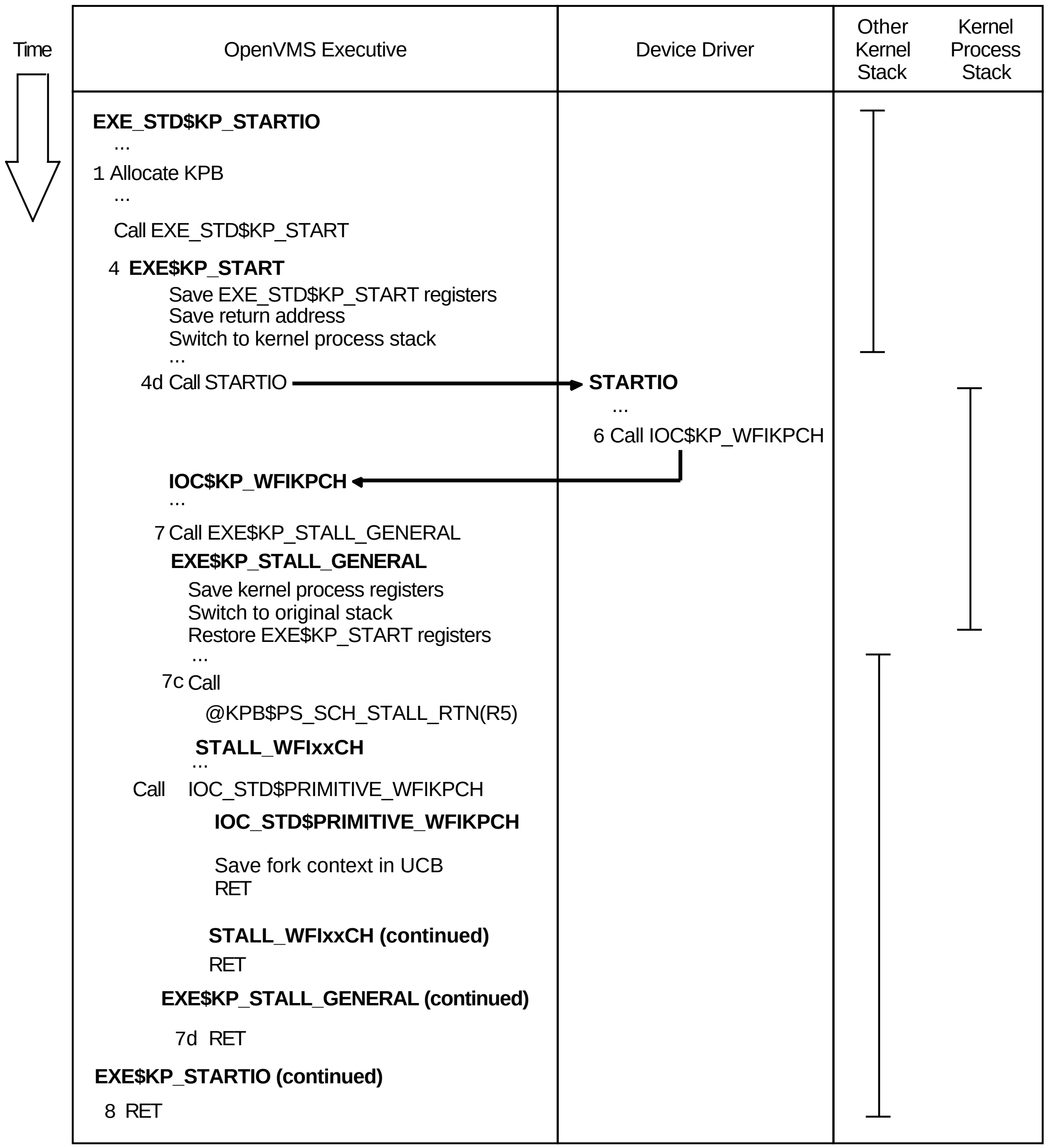
In Figures 3–2, 3–3, and 3–4, two barred lines appear in the rightmost column. Each represents the current stack of execution: either the kernel process private stack or a kernel stack.

3.2.7.1 Driver Kernel Process Startup

Figure 3–2 illustrates the flow of an I/O operation involving a driver kernel process from the creation of the kernel process to execute the start-I/O routine to the suspension of the kernel process to wait for a device interrupt. At the start of the process shown in the illustration, IOC$INITIATE has located the driver's start I/O routine and invokes it; in this example, it has issued a CALL to EXE_STD$KP_STARTIO, the routine identified by the DDTAB macro **start** argument.

Note that the numbers in Figure 3–2 refer to the numbers in the following description.

Figure 3–2 Driver Kernel Process Startup



EXE_STD$KP_STARTIO performs the following steps to create a kernel process thread of execution running the driver's start-I/O routine (STARTIO):[3]

1. It computes the kernel process required stack size as the larger of KPB$K_MIN_IO_STACK and DDT$IS_STACK_BCNT and calls EXE$KP_ALLOCATE_KPB to allocate a KPB and that much stack.
2. When EXE$KP_ALLOCATE_KPB returns a success status, it places the IRP and UCB addresses in KPB$PS_IRP and KPB$PS_UCB, respectively.

[3] This description focuses on those actions relevant to the control flow of a driver kernel process. Further details on the actions of EXE_STD$KP_STARTIO appear in *OpenVMS AXP Device Support: Reference*.

3. It performs a logical-OR of the value of DDT$IS_REG_MASK and the value of KPREG$K_MIN_IO_REG_MASK (which specifies R2 through R5, R12 through R15, and R26, R27, and R29), and excludes any registers that are illegal in a kernel process register save mask: R0, R1, R16 through R25, R27, R28, R30, and R31 (KPREG$K_ERR_REG_MASK). The result is a mask that includes only those registers that the kernel process support routines must save.
4. It calls EXE$KP_START. EXE$KP_START starts a driver kernel process thread of execution by taking the steps summarized in the following list:
   a. It saves the registers specified in the kernel process register save mask on the current stack.
   b. It saves the current stack pointer in KPB$PS_SAVED_SP.
   c. It switches to the kernel process private stack by loading SP from KPB$PS_STACK_BASE.
   d. It calls STARTIO, the procedure whose procedure value is in DDT$PS_KP_STARTIO, with the KPB address as the single argument.
5. STARTIO loads R3 and R5 from the IRP and UCB addresses in the KPB. It then acquires the device lock and initiates device activity.
6. After initiating device activity, STARTIO invokes the macro KP_STALL_WFIKPCH, which, for the given example, expands as shown in Example 3–3.

Example 3–3 Expansion of the KP_STALL_WFIKPCH Macro

```
;Expansion of KP_STALL_WFIKPCH DEVTMO,#6
                                  ;Assume top of stack contains IPL to
                                  ; be restored after wait has been
                                  ; set up
PUSHL    #6                       ;Timeout value
PUSHL    KPB                      ;KPB address
CALLS    #3,IOC$KP_WFIKPCH        ;
BLBC     R0,DEVTMO                ;If operation timed out,
                                  ; enter timeout routine
```

7. IOC$KP_WFIKPCH validates its arguments and copies them to the KPB. It records the procedure value of STALL_WFIXXCH in KPB$PS_SCH_STALL_RTN and calls EXE$KP_STALL_GENERAL to stall the kernel process.

   EXE$KP_STALL_GENERAL performs the following steps:[4]:
   a. It saves the kernel process context on the kernel process private stack.
   b. It restores the stack and register context that were current when the kernel process was entered.
   c. It calls STALL_WFIXXCH (the routine whose procedure value is in KPB$PS_SCH_STALL_RTN).

[4] This description focuses on those actions relevant to the control flow of a driver kernel process. Further details on the actions of EXE$KP_STALL_GENERAL appear in *OpenVMS AXP Device Support: Reference*.

STALL_WFIXXCH invokes the WFIKPCH macro, specifying the ENVIRONMENT=CALL parameter. The WFIKPCH macro invocation generates a standard call entry point in STALL_WFIXXCH and stores its procedure value in UCB$L_FPC. It then invokes IOC_STD$PRIMITIVE_WFIKPCH, which records the fork context of the driver kernel process, releases the device lock (restoring the IPL specified in the KP_STALL_WFIKPCH macro invocation), and returns to STALL_WFIXXCH. STALL_WFIXXCH returns to EXE$KP_STALL_GENERAL.

d. EXE$KP_STALL_GENERAL loads the success status SS$_NORMAL in R0 and returns to the routine whose return address was saved on the kernel stack, which, for this example, is EXE_STD$KP_STARTIO.

8. When control returns from EXE$KP_STALL_GENERAL, EXE_STD$KP_STARTIO tests the status in R0. If R0 contains a success status, EXE_STD$KP_STARTIO returns to its invoker, which, in this example, is IOC$INITIATE. If R0 contains an error, EXE$KP_START was unable to start the kernel process for some reason and EXE_STD$KP_STARTIO generates the fatal bugcheck INCONSTATE.

The control flow from IOC$INITIATE back to the $QIO requestor is the same as that for a driver that uses the simple fork process mechanism.

3.2.7.2 Resumption of a Driver Kernel Process by a Device Interrupt

Figure 3–3 illustrates the control flow from the time when the device activity completion interrupt resumes the driver kernel process to the time the driver completes servicing the interrupt.

Note that the numbers in Figure 3–3 refer to the numbers in the following description. Most of the details are left out of the steps here because they are detailed in *OpenVMS AXP Device Support: Reference*.

1. When the device interrupts, Alpha AXP Initiate Exception or Interrupt (IEI) Privileged Architecture Library code (PALcode) invokes IO_INTERRUPT.
2. IO_INTERRUPT calls the device's interrupt service routine (ISR).
3. At step 7c in Section 3.2.7.1, STALL_WFIXXCH invoked the WFIKPCH macro. The WFIKPCH macro invocation generated an entry point in STALL_WFIXXCH, and stored its procedure value in UCB$L_FPC. The device's interrupt service routine obtains the device lock and resumes STALL_WFIXXCH at this entry point by the following:

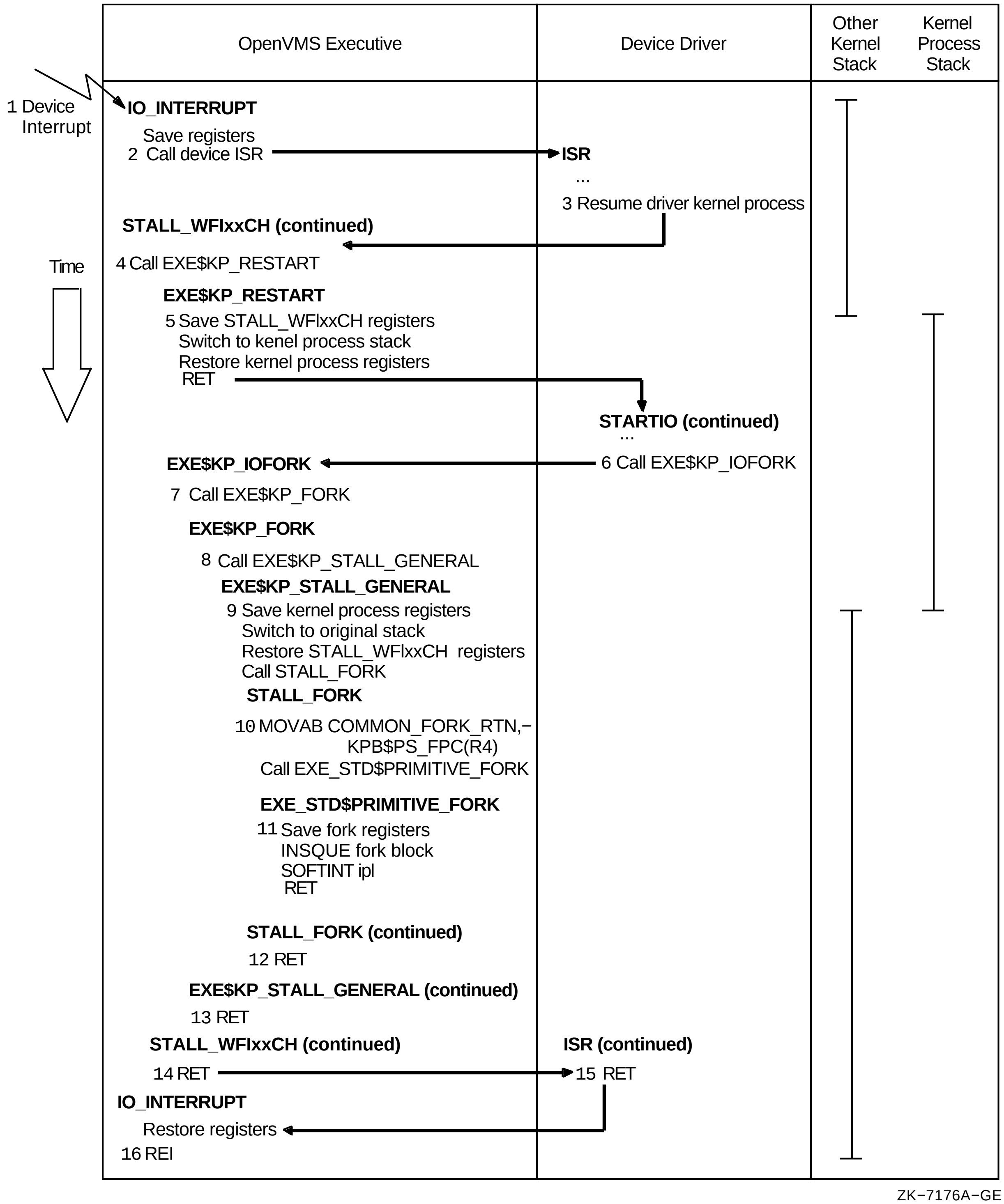
```
PUSHL   R5              ;Param3 = UCB address
PUSHL   UCB$Q_FR4(R5) ;Param2 = FR4 value
PUSHL   UCB$Q_FR3(R5) ;Param1 = FR3 value
CALLS   #3,@UCB$L_FPC(R5)
```

4. STALL_WFIXXCH calls EXE$KP_RESTART.

Note

A device driver can bypass this step and the overhead of an extra procedure call in its interrupt service routine if it can obtain the KPB address and call EXE$KP_RESTART directly as described in the previous step (Step 3).

Figure 3–3 Device Interrupt Resumes Driver Kernel Process



5. EXE$KP_RESTART saves the register context of its caller, switches to the kernel process private stack, and restores the kernel process registers. The most recent call frame on the kernel process private stack was left there when the driver kernel process earlier called IOC$KP_WFIKPCH. EXE$KP_

RESTART returns to the STARTIO procedure from its call to IOC$KP_WFIKPCH.

6. The STARTIO procedure performs device-specific status checks of the I/O operation that just completed. It performs only the steps that must be performed at device IPL, before invoking the KP_STALL_IOFORK macro to resume the kernel process at the lower fork IPL. The KP_STALL_IOFORK macro expands as follows:

```
PUSHL IRP$PS_KPB(R3)
CALLS #1,EXE$KP_IOFORK
```

7. EXE$KP_IOFORK clears UCB$V_TIM in UCB$L_STS to indicate that the device is no longer being timed for I/O and calls EXE$KP_FORK.
8. EXE$KP_FORK saves the kernel process fork context in the UCB fork block. It places the procedure value of STALL_FORK into KPB$PS_SCH_STALL_RTN and calls EXE$KP_STALL_GENERAL.
9. EXE$KP_STALL_GENERAL saves the kernel process register context in the KPB, switches to the original kernel stack and restores the registers that were saved in step 5, when the kernel process was resumed. It then calls STALL_FORK, the procedure whose procedure value is in KPB$PS_SCH_STALL_RTN.
10. STALL_FORK stores the procedure value of COMMON_FORK_RTN in KPB$PS_FPC, and invokes EXE_STD$PRIMITIVE_FORK.
11. EXE_STD$PRIMITIVE_FORK saves the fork parameters (which contain values previously in registers R3 and R4) in the UCB fork block, inserts the UCB fork block into the appropriate fork queue, requests a fork IPL interrupt if appropriate, and returns to STALL_FORK.
12. STALL_FORK returns to its caller, EXE$KP_STALL_GENERAL.
13. At this point, the most recent call frame on the original kernel stack is the one left there by STALL_WFIXXCH when it called EXE$KP_RESTART. EXE$KP_STALL_GENERAL returns to STALL_WFIXXCH.
14. STALL_WFIXXCH returns to the driver's interrupt service routine.
15. The interrupt service routine releases the device lock and returns to IO_INTERRUPT.
16. IO_INTERRUPT restores the registers it saved and dismisses the interrupt with a CALL_PAL REI instruction.

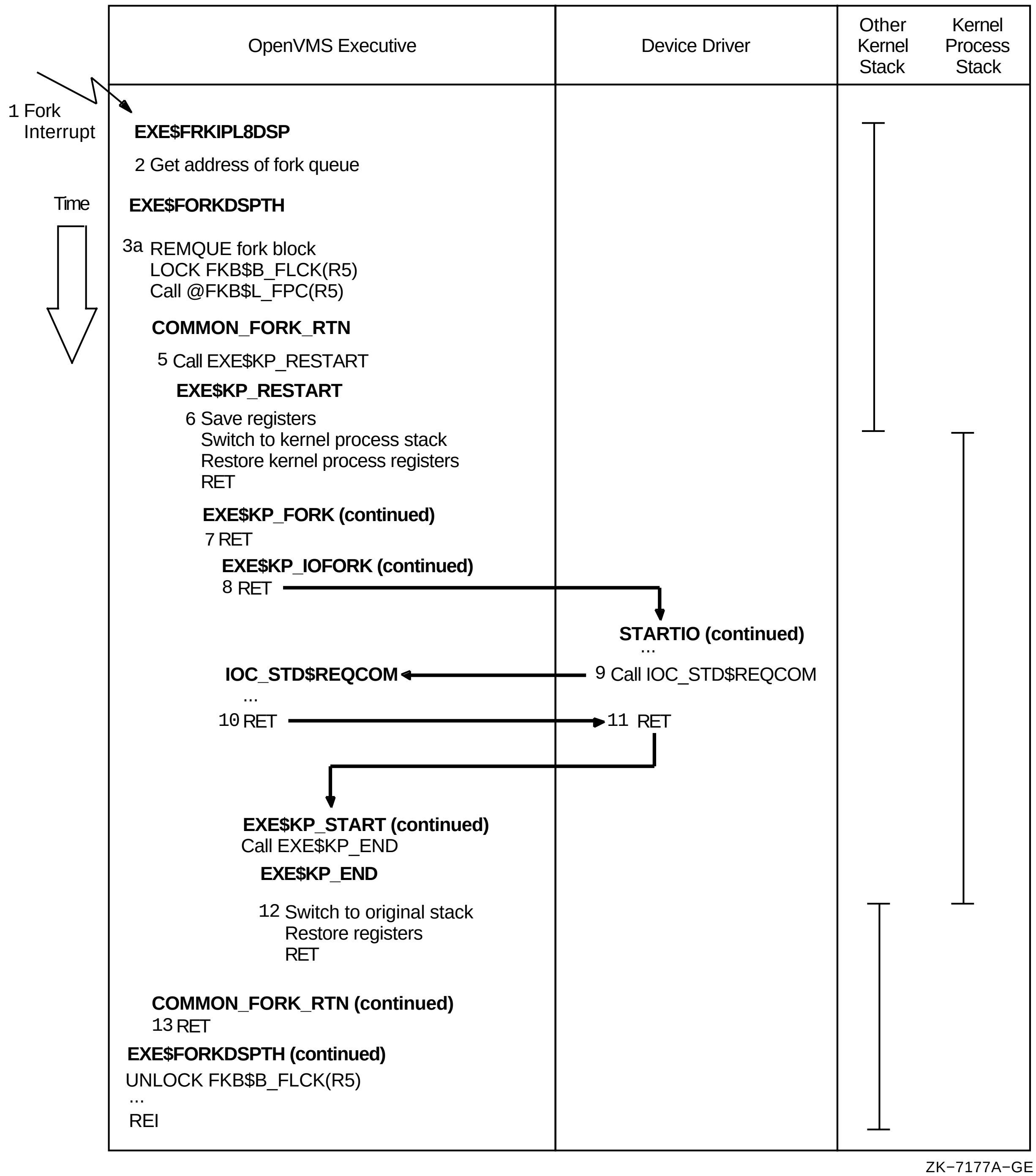
3.2.7.3 Resumption of a Driver Kernel Process by a Fork Interrupt

Figure 3–4 shows the control flow when the fork IPL software interrupt resumes the driver kernel process.

Note that the numbers in Figure 3–4 refer to the numbers in the following description. Most of the details are left out of the steps here because they are detailed in *OpenVMS AXP Device Support: Reference*.

1. When processor IPL drops below the fork IPL, the fork IPL software interrupt is granted. The fork dispatcher interrupt service routine, EXE$FRKIPL*x*DSP [where *x* is 6, 8, 9, 10, or 11, one of the fork IPLs] is entered. This example assumes a fork IPL of 8.

Figure 3–4 Fork Interrupt Resumes Driver Kernel Process



2. EXE$FRKIPL8DSP obtains the offset to the IPL 8 fork queue listhead and enters EXE$FORKDSPTH.

3. EXE$FORKDSPTH is a common entry point used by all fork IPL interrupt service routines. It resumes pending fork processes by performing the following steps:

   a. It removes a fork block from the fork queue. If no fork block was removed, it dismisses the fork IPL interrupt using the `CALL_PAL REI` instruction.

b. It acquires the fork lock whose index is in FKB$B_FLCK.

c. It resumes the fork process.

4. The fork process invokes COMMON_FORK_RTN.
5. COMMON_FORK_RTN calls EXE$KP_RESTART.
6. EXE$KP_RESTART saves the fork process register context on the current stack. R4 contains the KPB address of the kernel process that must be resumed. EXE$KP_RESTART switches to the kernel process private stack, restores the kernel process registers, and resumes the kernel process by executing the VAX MACRO instruction RET.

   The most recent call frame on the kernel process private stack is one left by EXE$KP_FORK when it earlier called EXE$KP_STALL_GENERAL. Thus the RET instruction resumes EXE$KP_FORK.
7. EXE$KP_FORK returns to its caller, EXE$KP_IOFORK.
8. EXE$KP_IOFORK returns to its caller, the STARTIO procedure.
9. The STARTIO procedure completes device-specific I/O postprocessing and invokes the KP_REQCOM macro. The KP_REQCOM macro expands to the following VAX MACRO instructions:

```
PUSHL  R5
PUSHL  R1
PUSHL  R6
CALLS  #3, IOC_STD$REQCOM
```

10. After IOC_STD$REQCOM performs the actions detailed in *OpenVMS AXP Device Support: Reference*, it returns to the STARTIO procedure.
11. At this point, the most recent call frame on the kernel process private stack is the one left there by EXE$KP_START when it earlier started up the driver kernel process and called the STARTIO procedure (see step 6d in Section 3.2.7.1. STARTIO returns to EXE$KP_START. EXE$KP_START calls EXE$KP_END to end the kernel process. If KPB$V_DEALLOC_AT_END is set in KPBIS_FLAGS, EXEKP_END calls EXE$KP_DEALLOCATE_KPB. EXE$KP_DEALLOCATE_KPB returns to EXE$KP_END.
12. At this point, the most recent call frame on the original kernel stack is the one left there by COMMON_FORK_RTN when it earlier called EXE$KP_RESTART. EXE$KP_END switches to the original kernel stack, restores registers that were saved by EXE$KP_RESTART, and returns to COMMON_FORK_RTN.
13. COMMON_FORK_RTN returns to EXE$FORKDSPTH, which releases the fork lock and proceeds to step 3a.

3.3 Mixing Fork and Kernel Processes

Ordinarily, a driver should use either the simple fork process or kernel process suspension mechanism exclusively. Doing so greatly simplifies comprehension of driver flow and maintenance of driver code.

It is possible for a driver to use the simple fork process mechanism for one execution thread and the kernel process mechanism for a different execution thread. Or, a single execution thread can use the simple fork process mechanism for certain tasks and later use the kernel process mechanism for others.

However, once a given driver thread has initiated a kernel process, the thread cannot use the simple fork mechanism until the kernel process has been terminated.

Warning

Attempting to perform a simple fork operation on a kernel process private stack will produce unpredictable if not disastrous results.

4

Allocating Map Registers and Other Counted Resources

Because AXP systems do not support the UNIBUS, Q22–bus, and MASSBUS adapters, the OpenVMS AXP operating system does not provide the following adapter-specific routines and macros that allocate and manage adapter map registers:

- IOC$ALOALTMAP, IOC$ALOALTMAPN, and IOC$ALOALTMAPSP
- IOC$ALOUBAMAP and IOC$ALOUBAMAPN
- IOC$LOADALTMAP (LOADALT macro)
- IOC$LOADMBAMAP (LOADMBA macro)
- IOC$LOADUBAMAP and IOC$LOADUBAMAPA (LOADUBA macro)
- IOC$RELALTMAP (RELALT macro)
- IOC$RELMAPREG (RELMPR macro)
- IOC$REQALTMAP (REQALT macro)
- IOC$REQMAPREG (REQMPR macro)

Instead, for AXP I/O subsystems that provide map registers, such as the TURBOchannel I/O processor for DEC 3000 AXP Model 500 systems, OpenVMS AXP provides a set of a routines that can manage the allocation of any resource that shares the following attributes of a set of map registers:

- The resource consists of an ordered set of items.
- The allocator can request one or more items. When requesting multiple items, the requester expects to receive a contiguous set of items. Thus, allocated items can be described by a starting number and a count.
- Allocation and deallocation of the resource are common operations and, thus, must be efficient and quick.
- A single deallocation may allow zero or more stalled allocation requests to proceed.

OpenVMS VAX systems record information relating to the availability and use of map registers in a set of arrays and fields within the adapter control block (ADP). OpenVMS AXP employs two new data structures for this purpose:

- A **counted resource allocation block** (CRAB), created by the OpenVMS adapter initialization routine, that describes a specific counted resource. The routine stores the address of the CRAB associated with a given adapter in ADP$L_CRAB.

Note

Code that needs to manage items of a private counted resource can use the system routines IOC$ALLOC_CRAB and IOC$DEALLOC_CRAB, described in *OpenVMS AXP Device Support: Reference*, to create a CRAB for that resource.

The number of resource items managed by a given CRAB is included in one of its fields. Resource items must be allocated in a numerically ordered, or contiguous series. A CRAB contains an array of quadword descriptors that record the location and length of a set of contiguous resource items that are free. Another CRAB field contains a value that is applied as a rounding factor to requests for resources to compute the actual number of items to be granted. For a detailed description of the CRAB, see *OpenVMS AXP Device Support: Reference.*

- A **counted resource context block** (CRCTX) that describes a specific request for a counted resource. The driver and the counted resource allocation routine exchange information in the CRCTX. A driver allocates a CRCTX before calling the counted resource allocation routine to obtain a certain number of items of the resource. For a detailed description of the CRCTX, see *OpenVMS AXP Device Support: Reference.*

Despite the new structures and new routines, an OpenVMS AXP device driver performs most of the same tasks as an OpenVMS VAX device driver when setting up and completing a direct memory access (DMA) transfer. An OpenVMS AXP device driver:

1. Calls IOC$ALLOC_CRCTX to obtain a CRCTX that describes a request for map registers
2. Loads the request count into the CRCTX$L_ITEM_CNT field
3. Calls IOC$ALLOC_CNT_RES to request the map registers
4. Calls IOC$LOAD_MAP to load the map registers granted in the allocation request
5. Prepares device registers for the transfer and activates the device
6. Calls IOC$DEALLOC_CNT_RES to free the registers for use by other requesters
7. Calls IOC$DEALLOC_CRCTX to deallocate the CRCTX

The following sections describe these steps.

4.1 Allocating a Counted Resource Context Block

A driver calls IOC$ALLOC_CRCTX to allocate and initialize a counted resource context block (CRCTX). The CRCTX describes a specific request for a given counted resource, such as a set of map registers. The driver subsequently uses the CRCTX as input to IOC$ALLOC_CNT_RES to allocate a set of the items managed as a counted resource.

IOC$ALLOC_CRCTX requires as input the address of the CRAB that describes the counted resource. For adapters that provide a counted resource, such as a set of map registers, ADP$L_CRAB contains this address.

The following example illustrates a call to IOC$ALLOC_CRCTX that returns the address of the allocated CRCTX to UCB$L_CRCTX, a field in an extended UCB:

```
70$:    PUSHAL  UCB$L_CRCTX(R5)         ; Pass cell to receive CRCTX address
        PUSHL   ADP$L_CRAB(R1)          ; Pass CRAB as argument
        CALLS   #2,IOC$ALLOC_CRCTX      ; Initialize the CRCTX
        BLBC    R0,200$                 ; Branch if failure status returned
```

To avoid the overhead of allocating (and deallocating) a CRCTX for each DMA transfer, drivers often obtain multiple CRCTXs in their controller or unit initialization routines, linking them from a data structure such as the UCB so that they will be available for later use.

See *OpenVMS AXP Device Support: Reference* for a detailed description of IOC$ALLOC_CRCTX.

4.2 Allocating Counted Resource Items

A driver calls IOC$ALLOC_CNT_RES to allocate a requested number of items from a counted resource. IOC$ALLOC_CNT_RES requires the addresses of both the CRAB and the CRCTX as input parameters. The resource request is described in the CRCTX structure; the counted resource itself is described in the CRAB.

A driver typically initializes the following fields of the CRCTX before calling IOC$ALLOC_CNT_RES.

Field	Description
CRCTX$L_ITEM_CNT	Number of items to be allocated. When requesting map registers, this value in this field should include two extra map registers to be allocated and loaded as a guard page to prevent runaway transfers. There may be additional bus-specific requirements. See *OpenVMS AXP Device Support: Developer's Guide*.
CRCTX$L_CALLBACK	Procedure value of the callback routine to be called when the deallocation of resource items allows a stalled resource request to be granted. A value of 0 in this field indicates that, on an allocation failure, control should return to the caller immediately without queuing the CRCTX to the CRAM's wait queue.

A caller can also specify the upper and lower bounds of the search for allocatable resource items by supplying values for CRCTX$L_LOW_BOUND and CRCTX$L_UP_BOUND.

IOC$ALLOC_CNT_RES always returns to its caller immediately, whether the allocation request is granted immediately, is stalled, or is unsuccessful. If the request is granted immediately, or when a stalled request is eventually granted, IOC$ALLOC_CNT_RES returns the number of the first item granted to the caller in CRCTX$L_ITEM_NUM and sets CRCTX$V_ITEM_VALID in CRCTX$L_FLAGS.

If there are waiters for the counted resource, or if there are insufficient resource items to satisfy the request, IOC$ALLOC_CNT_RES saves the current values of R3, R4, and R5 in the CRCTX fork block. IOC$ALLOC_CNT_RES writes a –1 to CRCTX$L_ITEM_NUM, and inserts the CRCTX in the resource-wait queue (headed by CRAB$L_WQFL). It then returns SS$_INSFMAPREG status to its caller.

Note

If a counted resource request does not specify a callback routine (CRCTX$L_CALLBACK), IOC$ALLOC_CNT_RES does not insert its CRCTX in the resource-wait queue. Rather, it returns SS$_INSFMAPREG status to its caller.

A driver must not deallocate the CRCTX while the resource request it describes is stalled by IOC$ALLOC_CNT_RES. (If the driver must cancel the allocation request, it should call IOC$CANCEL_CNT_RES.)

When a counted resource deallocation occurs, the first CRCTX is removed from the resource-wait queue and the allocation is attempted again. If IOC$ALLOC_CNT_RES is now able to grant the requested number of resource items, it issues a JSB to the callback routine (CRCTX$L_CALLBACK), passing it the following values:

Location	Contents
R0	SS$_NORMAL
R1	Address of CRAB
R2	Address of CRCTX
R3	Contents of R3 at the time of the original allocation request (CRCTX$Q_FR3)
R4	Contents of R4 at the time of the original allocation request (CRCTX$Q_FR4)
R5	Contents of R5 at the time of the original allocation request (CRCTX$Q_FR5)
Other registers	Destroyed

The callback routine checks R0 to determine whether it has been called with SS$_NORMAL or SS$_CANCEL status (from IOC$CANCEL_CNT_RES). If the former, the routine typically proceeds to loads the map registers that have been allocated. The callback routine must preserve all registers it uses other than R0 through R5 and exit with an RSB instruction.

The following example illustrates a call to IOC$ALLOC_CNT_RES:

```
40$:    MOVL    SCDRP$L_BOFF(R5),R0     ; Get byte offset
        ADDL    SCDRP$L_BCNT(R5),R0     ; Add in byte count
        ADDL    G^MMG$GL_BWP_MASK,R0    ; Round up to number of pages
        ADDL    G^MMG$GL_PAGE_SIZE,R0   ; Add extra "no access" page
        ASHL    G^MMG$GL_VA_TO_VPN,R0,- ; Get number of pages involved
                   CRCTX$L_ITEM_CNT(R2) ; Pass as number of contiguous
                                        ; registers to allocate
        MOVAB   G^SCS$MAP_RETRY,-       ; SCS$MAP_RETRY is callback routine
                   CRCTX$L_CALLBACK(R2)
        PUSHL   R2                      ; Push CRCTX as argument
        PUSHL   ADP$L_CRAB(R4)          ; Push CRAB as argument
        CALLS   #2,IOC$ALLOC_CNT_RES    ; Allocate the map registers
        BLBC    R0,110$                 ; If allocation is not successful,
                                        ; branch; otherwise proceed
                                        ; to load map registers
        .
        .
        .
110$:   CMPL    #SS$_INSFMAPREG,R0      ; INSFMAPREG means request queued
        BNEQ    120$                    ; Other status means error; branch
        MOVL    #_C_MAP_ALLOC_WAIT_STATE,-  ; Record wait state in
                   CDRP$L_WAIT_STATE(R5)    ; CDRP
        MOVL    #SS$_INSFMAP,R0         ; Return status to caller of this
                                        ; driver routine
        RSB
120$:   ; Process returned errors (other than SS$_INSFMAPREG)
```

The OpenVMS AXP operating system allows you to indicate that a counted resource request should take precedence over any waiting request by setting the CRCTX$V_HIGH_PRIO bit in CRCTX$L_FLAGS. A driver employs a high-priority counted resource request to preempt normal I/O activity and service some exception condition from the device. (For instance, during a multivolume backup, a tape driver might make a high-priority request, when it encounters the end-of-tape (EOT) marker, to get a subsequent tape loaded before normal I/O activity to the tape can resume. A disk driver might issue a high-priority request to service a disk offline condition.)

IOC$ALLOC_CNT_RES never stalls a high-priority counted resource request or places its CRCTX in a resource-wait queue. Rather, it attempts to allocate the requested number of resource items immediately. If IOC$ALLOC_CNT_RES cannot grant the requested number of items, it returns SS$_INSFMAPREG status to its caller.

See *OpenVMS AXP Device Support: Reference* for a detailed description of IOC$ALLOC_CNT_RES.

4.3 Loading Map Registers

A driver calls IOC$LOAD_MAP to load a set of adapter-specific map registers. The driver must have previously allocated the map registers (including an extra two to serve as a guard page) in calls to IOC$ALLOC_CRCTX and IOC$ALLOC_CNT_RES.

IOC$LOAD_MAP requires the following as input:

- the address of the ADP of the adapter that provides the map registers
- the address of the CRCTX that describes the map register allocation
- the system virtual address of the page table entry (PTE) for the first page to be used in the DMA transfer

- the Byte offset into the first page of the transfer

IOC$LOAD_MAP returns a specified location a port-specific address of a DMA buffer.

The following example illustrates a call to IOC$LOAD_MAP:

```
100$:   PUSHAL  UCB$L_ARG(R4)           ; Cell for returned DMA address
        MOVZWL  BD$W_PAGE_OFFSET(R3),-(SP) ; Pass starting buffer offset
        PUSHL   BD$L_SVAPTE(R3)         ; Pass SVAPTE as argument
        PUSHL   R2                      ; Pass CRCTX as argument
        PUSHL   PDT$L_ADP(R4)           ; Pass ADP as argument
        CALLS   #5,IOC$LOAD_MAP         ; Load the allocated map registers
```

See *OpenVMS AXP Device Support: Reference* for a detailed description of IOC$LOAD_MAP.

Having loaded the map registers for a DMA transfer, a driver typically performs some of the following steps to initiate the transfer:

- Loads the port-specific DMA address into a device DMA address register. Some manipulation of the address value might be needed, depending upon the hardware. (For instance, a DEC 3000 AXP Model 500 driver must clear the two low bits before writing to the register.)
- Computes the transfer length and loads a device transfer count register. Typically a driver derives the transfer length from a field such as UCB$L_BCNT.
- Sets to GO byte in the device CSR (possibly indicating the direction of the transfer as well) by writing a mask to the CSR.

4.4 Deallocating a Number of Counted Resources

A driver calls IOC$DEALLOC_CNT_RES to deallocate a requested number of items of a counted resource. IOC$DEALLOC_CNT_RES requires the addresses of both the CRAB and CRCTX as input. After deallocating the items, IOC$DEALLOC_CNT_RES attempts to restart any waiters for the resource.

The following example illustrates a call to IOC$DEALLOC_CNT_RES:

```
        PUSHL   R2                      ; Push CRCTX as argument
        PUSHL   ADP$L_CRAB(R4)          ; Push CRAB as argument
        CALLS   #2,IOC$DEALLOC_CNT_RES  ; Deallocate the map registers
```

See *OpenVMS AXP Device Support: Reference* for a detailed description of IOC$DEALLOC_CNT_RES.

4.5 Deallocating a Counted Resource Context Block

A driver calls IOC$DEALLOC_CRCTX to deallocate a CRCTX. IOC$DEALLOC_CRCTX requires only the address of the CRCTX as input.

A driver must not deallocate a CRCTX that describes a request that has been stalled waiting for sufficient resource items to be made available (that is, a CRCTX that is in a given CRAB wait queue). Prior to deallocating such a CRCTX, a driver should call IOC$CANCEL_CNT_RES to cancel the resource request.

The following example illustrates a call to IOC$DEALLOC_CRCTX:

```
PUSHL   R2                      ; Pass CRCTX as argument
CALLS   #1,IOC$DEALLOC_CRCTX    ; Deallocate the CRCTX
```

See *OpenVMS AXP Device Support: Reference* for a detailed description of IOC$DEALLOC_CRCTX.

5

Synchronization Requirements for OpenVMS AXP Device Drivers

This chapter discusses special synchronization requirements for OpenVMS AXP device drivers beyond the basic synchronization requirements for OpenVMS AXP device drivers discussed in the *OpenVMS AXP Device Support: Developer's Guide*. It focuses on the following areas:

- Section 5.1 describes why and how you must use OpenVMS driver multiprocessing synchronization semantics when creating an OpenVMS AXP device driver.
- Section 5.2 discusses why it is important to identify driver operations that depend on the exact ordering of reads and writes to memory and shows how to enforce this ordering.
- Section 5.3 explains how VAX systems and AXP systems differ in their ability to access, without interruption, byte-, word-, and longword-sized data items, and suggests ways of overcoming these differences to synchronize access to such items.
- Section 5.4 describes how to synchronize different instruction streams on an OpenVMS AXP system.

5.1 Producing a Multiprocessing-Ready Driver

All OpenVMS AXP device drivers must adhere to the rules for OpenVMS multiprocessing device drivers as listed in the multiprocessing requirements appendix of the *OpenVMS AXP Device Support: Developer's Guide*.

The following is a general summary of those rules for OpenVMS AXP device drivers:

- Specify **smp**=**YES** in the DPTAB macro invocation.
- Use the following spin lock synchronization macros instead of macros that simply raise and lower IPL:
  - FORKLOCK/FORKUNLOCK
  - DEVICELOCK/DEVICEUNLOCK
  - LOCK/UNLOCK

  Note that the **lockipl** argument of these macros is ignored on OpenVMS AXP systems. The operating system automatically obtains the lock's IPL from the spin lock or fork lock data structure, or from the spin lock IPL vector.
- Initialize field FKB$B_FLCK of each fork block with the index of the fork lock that synchronizes access to the structure in which the fork block resides. Typically, drivers initialize the UCB fork block by issuing a DPT_STORE macro within a DPTAB macro invocation.

Note that you can no longer store a fork IPL in this field; the field's alias, UCB$B_FIPL, has been deleted.

5.2 Enforcing the Order of Reads and Writes

VAX multiprocessing systems have traditionally been designed so that if one processor in the 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 and then writes a flag, CPU B can determine that the data buffer has changed by examining the value of the flag.

OpenVMS AXP systems may reorder read and write operations to memory 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 writes 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.

Device driver threads that share data in multiprocessing environments or with DMA I/O devices must be careful to insert an Alpha AXP Memory Barrier (MB) instruction as appropriate, before and after data references. The MB instruction guarantees that all subsequent loads or stores will not access memory until after all previous loads and stores have accessed memory, as observed by other processors.

For traditional, common device driver operations, you can rely on OpenVMS system routines that initiate DMA device operations to memory or that acquire spin locks that protect specific system databases in a multiprocessing system to insert the required memory barriers. The following are some examples of how OpenVMS AXP provides memory barriers transparently when needed to properly order memory operations involving device drivers:

- When a driver is writing a buffer to a disk (involving a device that performs a DMA read operation to memory), an MB instruction must be issued before the driver initiates the write transaction and the device must issue an MB instruction after receiving the start signal but before starting the DMA read. A driver normally calls the system routine IOC$CRAM_IO (or IOC$CRAM_QUEUE and IOC$CRAM_WAIT) to deliver data and the start command to the DMA device's registers. Because these routines issue the appropriate MB instructions on behalf of the driver, the driver need not include an explicit memory barrier.
- When a DMA I/O device has written data to memory (for instance, paging in a page from disk), the DMA device must issue an MB instruction before posting a completion interrupt, and the OpenVMS I/O interrupt dispatcher (IO_INTERRUPT) issues an MB instruction to guarantee that the data is visible to the interrupted processor before invoking the driver's interrupt service routine.
- All routines and macros that acquire spin locks, fork locks, and device locks to synchronize access to a specific database in a multiprocessing system issue an MB instruction prior to obtaining the lock.

Note

The uniprocessing versions of the spin lock routines and macros do not provide memory barriers.

There are two ways to generate an MB instruction from VAX MACRO code:

- The MACRO-32 compiler for OpenVMS AXP generates an implicit memory barrier when processing any of the VAX interlocked instructions (such as BBSSI, BBCCI, and ADAWI) and interlocked queue instructions.
- The MACRO-32 compiler provides the EVAX_MB built-in to generate an explicit memory barrier.

There are certain instances when a driver must include an explicit memory barrier. For instance, if a driver and a device controller exchange data and effect transactions by means of some in-memory structure, such as a command buffer and a doorbell register, a driver ordinarily does not use IOC$CRAM_IO or IOC$CRAM_QUEUE after setting up device registers with the appropriate memory addresses. In such a case, a driver must take care to explicitly order the writes to the command buffer and the write to the doorbell register to enforce the order of reads and writes involving the buffer. The MACRO-32 compiler for OpenVMS AXP provides an EVAX_MB built-in to allow you to insert a memory barrier prior to the latter write, as in the following example:

```
; Set up the SCSI base register with command ring's physical address
;-
        MOVL    SPDT$PS_CMD_RING(R4),R2 ; Get the SVA of command ring
        BSBW    GET_PHY_ADDR            ; Convert it to physical address
        DEVICELOCK -                    ; Get device lock and raise IPL
           LOCKADDR=SPDT$L_DLCK(R4),-
           LOCKIPL=SPDT$B_DIPL(R4),-
           SAVIPL=-(SP),-
           PRESERVE=NO
        MOVL    SPDT$PS_SCSI_BASE(R4),R0 ; Get address of SCSI base register
        EVAX_STQ  R1,(R0)               ; Write cmd ring addr. to SCSI base register
        EVAX_MB                         ; Do memory barrier for correct instr. sequence
        MOVL    SPDT$PS_SCSI_DB(R4),R0  ; Get address of SCSI doorbell register
        EVAX_STQ  R1,(R0)               ; Ring the SCSI doorbell register
```

5.3 Ensuring Synchronized Access of Byte-, Word-, and Longword-Sized Data Items

The VAX architecture supports instructions that can read or write byte- and word-sized data in a single noninterruptible operation. The Alpha AXP architecture supports instructions that read or write longword- and quadword-sized data uninterruptedly. Because the AXP instruction sequence simplythat accomplishes byte- and word-sized reads is interruptible, operations on byte and word data that are automatic on VAX systems, are no longer atomic on AXP systems.

In addition, this difference in the granularity of memory access can also affect the definition of which data is shared. On VAX systems, a byte- or word-sized item that is shared can be manipulated without regard to neighboring data. On AXP systems, the entire longword or quadword that contains the byte- or word-sized item must be manipulated. If a word-sized (or longword-sized) item crosses a longword- or quadword-address boundary, two longwords or quadwords may be manipulated. Thus, because of its proximity to an explicitly shared data item, neighboring data may become *unintentionally* shared.

A device driver must take steps beyond those required in traditional interrupt priority level (IPL) and spin lock synchronization to ensure that bytes, words, and longwords are accessed without interference. Although interlocked instructions (BBSSI, BBCCI, and ADAWI) generate memory barriers and interlocked OpenVMS AXP code sequences, they assume a byte granularity environment. Where the data segment on which these and other instructions operate may be concurrently written by different threads, you may need to impose additional synchronization as follows:

- Align data structures on natural address boundaries in memory. That is, align all fields on a natural boundary: bytes at any byte address, words at any address that is a multiple of 2, longwords at any address that is a multiple of 4, and quadwords at any address that is a multiple of 8.
- Inspect shared fields and fields around them for intralongword or intraquadword granularity problems. For instance, identify word and byte fields that are shared between threads running at different IPLs—for instance, a UCB bitmask where bits are accessed at device IPL and fork IPL or a UCB quadword that consists of a longword accessed at IPL$_ASTDEL and a word accessed at fork IPL.

  Resolve intralongword and intraquadword granularity problems by padding the bytes, words, or longwords involved, or promoting them to longword or quadword fields. A bit that is changed by BBSSI or BBCCI, or a word modified by ADAWI, should reside in a longword where the other portions of the longword are not modified by an independent and concurrent instruction thread. A longword bitmask should contain bits accessed only at fork IPL or at device IPL, not at both.
- Identify base structure alignment to the MACRO-32 compiler, so that the MACRO compiler can generate the most optimal and safest instruction sequence to access its fields. For instance, if you know that the base alignment of a structure is at a longword boundary, use the following:

  ```
  .SYMBOL_ALIGNMENT LONG
  ```

  ```
  .SYMBOL_ALIGNMENT QUAD
  ```

  Whenever the MACRO-32 compiler encounters a reference in which a symbol that is defined in the context of one of these directives is used as an offset from a register, it generates Alpha AXP instructions reflecting the specified symbol alignment and its own register alignment assumptions. Note that, when you use one of these directives, you must insert the following directive in the data declarations when the specified symbol alignment is no longer in effect:

  ```
  .SYMBOL_ALIGNMENT NONE
  ```

  Note

  The .SYMBOL_ALIGNMENT directive does not work in the context of the $DEFINI, $DEF, _VIELD, and $DEFEND macros.

  See *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code* for additional information on MACRO-32 compiler alignment assumptions and instructions for using the .SYMBOL_ALIGNMENT directive.

5.4 Using Instruction Memory Barriers

Code that modifies the instruction stream must be changed to properly synchronize the old and new instructions streams. Use of an RET instruction to accomplish this will not work on OpenVMS AXP systems.

If a driver code sequence changes the expected instruction stream, it must issue an Instruction Memory Barrier (IMB) instruction after changing the instruction stream and before the time the change is executed. For example, if a driver stores an instruction sequence in an extension to the unit control block (UCB) and then transfers control there, it must issue an IMB instruction after storing the data in the UCB but before transferring control to the UCB data.

The MACRO-32 compiler for OpenVMS AXP provides the EVAX_IMB built-in to explicitly insert an IMB instruction in the instruction stream.

6
Conversion Guidelines

This chapter describes the tasks required to convert an OpenVMS VAX device driver to an OpenVMS AXP Step 2 device driver. For more details about the macros, system routines, and entry points listed in this chapter, see *OpenVMS AXP Device Support: Reference*. For more details about porting VAX MACRO code to OpenVMS AXP, see *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code*.

6.1 OpenVMS AXP Device Driver Program Sections

An OpenVMS AXP device driver consists of three distinct program sections, or **psects**:

- $$$105_PROLOGUE, which contains the DPT and is defined automatically by the DPTAB macro.
- $$$110_DATA, which contains driver data such as the driver dispatch table (DDT) and the function decision table (FDT)
- $$$115_DRIVER, which contains driver code

Because OpenVMS AXP compiler technology does not allow code and data to reside together in the same psect, you must keep code and data in the proper psects of an OpenVMS AXP driver. Moreover, because OpenVMS AXP drivers are loadable executive images, you must ensure that the psect attributes are correctly and consistently defined so as to allow the image to be linked properly.

The following are guidelines for psect declaration:

- Add an invocation of the DRIVER_CODE macro prior to the first line of executable code in the driver. By default, the DRIVER_CODE macro declares the psect $$$115_DRIVER. However, you can specify any alternative psect name consistent with the naming and linking conventions of the OpenVMS VAX driver you are porting to OpenVMS AXP.

  Unlike its behavior in OpenVMS VAX device drivers, the DDTAB macro does not define the $$$115_DRIVER psect for OpenVMS AXP device drivers. Rather it defines the data psect ($110_DATA) in which the DDT resides.
- OpenVMS macros that construct data, such as DDTAB and FUNCTAB, automatically invoke the DRIVER_DATA macro prior to creating the data. By default, the DRIVER_DATA macro declares the psect $$$110_DATA.
- You must move all driver-specific data structures currently defined within the body of the code (in psect $$$115_DRIVER) to a data psect. Although the DRIVER_DATA macro declares the psect $$$110_DATA by default, you can specify any alternative psect name consistent with the naming and linking conventions of the OpenVMS VAX driver you are porting to OpenVMS AXP.

- If the driver consists of multiple source modules, you should replace each explicit setting of the $$$115_DRIVER psect with an invocation of the DRIVER_CODE macro to ensure that the correct standard psect for driver code sections is always used.

6.2 DPTAB Changes

The driver prologue table (DPT) must declare that the driver is a Step 2 driver. To identify an OpenVMS AXP Step 2 driver, specify **step**=**2** when invoking the DPTAB macro. The macro creates the constant DPT$K_STEP_2 and inserts it into the DPT$IW_STEP field of the driver prologue table (DPT). The macro also inserts the value DPT$K_STEP2_V2 in the DPT$IW_STEPVER field.

If you do not make this change, compilation errors will result. OpenVMS AXP uses the value in DPT$IW_STEP to detect driver sources that have not been modified to conform to the currently supported OpenVMS AXP driver implementation. OpenVMS AXP uses the value in DPT$IW_STEPVER to enforce the most recent driver loading procedure requirements.

In an OpenVMS VAX driver, the DPT must be at the very beginning of the driver image. In an OpenVMS AXP driver, the DPT can be in any read/write image section of the driver.

See *OpenVMS AXP Device Support: Reference* for more information about the DPT and the DPTAB macro.

6.3 DDTAB Changes

The following sections summarize DDTAB macro changes you must make when converting an OpenVMS VAX driver to an OpenVMS AXP driver.

6.3.1 DDTAB Routine Name Changes

The routines pointed to by the driver dispatch table (DDT) must conform to Step 2 requirements. You must add entry point declarations for driver-specific routines, but the names may remain unchanged. Change any OpenVMS routine name referenced in the driver's DDTAB macro invocation as follows:

1. Replace **cancel**=IOC$CANCELIO with **cancel**=IOC_STD$CANCELIO.
2. Replace **mntver**=IOC$MNTVER with **mntver**=IOC_STD$MNTVER.

See *OpenVMS AXP Device Support: Reference* for more information about the driver dispatch table (DDT) and the DDTAB macro.

6.3.2 Specifying Controller and Unit Initialization Routines

An OpenVMS VAX device driver specifies the location of its controller initialization routine by issuing a DPT_STORE macro of the following form:

```
DPT_STORE  CRB, CRB$L_INTD+VEC$L_INITIAL, D, XX_CTRL_INIT
```

Similarly, an OpenVMS VAX driver may specify the location of its unit initialization routine using the following:

```
DPT_STORE  CRB, CRB$L_INTD+VEC$L_UNITINIT, D, XX_UNIT_INIT
```

An OpenVMS AXP device driver must use the **ctrlinit** and **unitinit** arguments to the DDTAB macro to specify the controller initialization routine address:

```
DDTAB   -
        ctrlinit=XX_CTRL_INIT,-
        unitinit=XX_UNIT_INIT,-
        .
        .
        .
```

See *OpenVMS AXP Device Support: Reference* for a description of the DDTAB macro.

6.3.3 Simple Fork Mechanism—JSB-Based Fork Routines

Chapter 3 describes alternatives available to OpenVMS AXP device drivers for suspension of execution. If you want to continue using the simple fork mechanism with JSB-based fork routines for the code path from start I/O through request complete, you must use the DDTAB JSB_START parameter to identify your start I/O routine:

```
DDTAB   -
        JSB_START = driver_startio_routine
```

instead of:

```
DDTAB   -
        START   = driver_startio_routine
```

By doing so, the IOC$START_C2J CALL-to-JSB jacket routine is actually used as the start I/O entry. The IOC$START_C2J routine invokes the routine specified by the JSB_START parameter. A similar approach can also be used for the alternate start I/O entry point. The DDTAB JSB_ALTSTART parameter is used to specify the alternate start I/O entry:

```
DDTAB   -
        JSB_ALTSTART = driver_altstart_routine
```

instead of:

```
DDTAB   -
        ALTSTART = driver_altstart_routine
```

The performance cost of this approach is one additional level of routine call to dispatch an IRP to the driver's start I/O routine or alternate start I/O routine.

6.3.4 Kernel Process Mechanism

If you want to use the kernel process mechanism, you must use the DDTAB KP_STARTIO parameter to identify your start I/O routine as follows:

```
DDTAB   -
        START   = EXE_STD$KP_STARTIO,-
        KP_STARTIO = driver_startio_routine
```

6.4 Specifying an Interrupt Service Routine

An OpenVMS VAX device driver specifies the location of an interrupt service routine by issuing a DPT_STORE macro of the following form:

```
DPT_STORE  CRB, CRB$L_INTD+VEC$L_ISR, D, XX_ISR
```

An OpenVMS AXP device driver specifies the location of an interrupt service routine by issuing the new DPT_STORE_ISR macro, as follows:

```
DPT_STORE_ISR  CRB$L_INTD, XX_ISR
```

See *OpenVMS AXP Device Support: Reference* for a description of the DPT_STORE_ISR macro.

6.5 Interrupt Service Routine Entry Points

The interrupt service routine in an OpenVMS AXP device driver is a standard call interface routine. The interrupt service routine is invoked by the system service dispatcher with two parameters: the address of the IDB and the SCB vector offset.

The .CALL_ENTRY or .ENTRY directives must be used to identify the entry point of an OpenVMS AXP Step 2 device driver. The interrupt service routine should save and restore any non-scratch register that it uses and it must transfer control back to the interrupt dispatcher via a RET instruction. For example:

```
MY_ISR: .CALL_ENTRY PRESERVE=<R2,R3,R4,R5>
        MOVL    4(AP),R4    ; retrieve IDB address
        .
        .
        .
        RET                 ; return back to interrupt dispatch
```

In contrast, an OpenVMS VAX interrupt service routine is not a standard call procedure. It exits and dismisses the interrupt via an REI instruction.

6.6 Start I/O and Alternate Start I/O Entry Points

Section 3.2 describes the use of the kernel process services for the code path from start I/O through request complete. The entry point of a kernel process start I/O routine should be identified using either the .CALL_ENTRY or .ENTRY directives as follows:

```
MY_STARTIO:
   .CALL_ENTRY
```

Section 3.2.2 describes the complete requirements for a kernel process start I/O routine.

If you choose to continue to use the simple fork mechanism, you must choose between using a JSB-based fork routine environment that is very similar to the OpenVMS VAX fork environment and a standard call based fork environment. Section 3.1 describes the differences between the OpenVMS VAX and OpenVMS AXP fork mechanisms.

The code path from start I/O through request complete in some existing drivers written in MACRO-32 may be difficult and error prone to convert to the standard call fork interfaces. This can apply to complex drivers that make extensive use of branches between routines within the same module. If you choose to continue to use the JSB-based environment, you should place the following entry point directives at the beginning of your start I/O and alternate start I/O routines:

```
MY_STARTIO:
  .JSB_ENTRY INPUT=<R3,R5>,SCRATCH=<R0,R1,R2,R3,R4>
```

If you choose to convert your start I/O code path to the new standard call interface, you should use the $DRIVER_START_ENTRY and $DRIVER_ALTSTART_ENTRY macros to identify the entry points of your start I/O and alternate start I/O routines:

```
MY_STARTIO:
   $DRIVER_START_ENTRY
```

For information about additional requirements and guidelines for using the standard call environment for fork routines, see Section 7.4.

6.7 Using the Driver Entry Point Routine Call Interfaces

To use the call interfaces required for Step 2 driver-supplied routines, perform the following tasks:

1. Use the appropriate macro to identify entry points in your driver. Step 2 driver entry point macros include the following:
   - $DRIVER_CANCEL_ENTRY
   - $DRIVER_CANCEL_SELECTIVE_ENTRY
   - $DRIVER_CHANNEL_ASSIGN_ENTRY
   - $DRIVER_CLONEDUCB_ENTRY
   - $DRIVER_CTRLINIT_ENTRY
   - $DRIVER_ERRRTN_ENTRY
   - $DRIVER_FDT_ENTRY
   - $DRIVER_MNTVER_ENTRY
   - $DRIVER_REGDUMP_ENTRY
   - $DRIVER_DELIVER_ENTRY
   - $DRIVER_UNITINIT_ENTRY
2. Use the default **F ETCH=YES** parameter value.

   This value causes the standard interface parameters to be fetched and copied to their OpenVMS VAX JSB interface registers, for example:

   ```
   $DRIVER_UNITINIT_ENTRY FETCH=YES
   ```

   results in

   ```
   MOVL  #SS$_NORMAL,R0
   MOVL  UNITARG$_IDB(AP),R4
   MOVL  UNITARG$_UCB(AP),R5
   ```
3. Use the default **PRESERVE** parameter value.

   The default is the set of registers that was allowed to be scratched by the OpenVMS VAX JSB interface routine, for example:

   ```
   $DRIVER_UNITINIT_ENTRY
   ```

   results in

   PRESERVE=<R2>

   This set of registers is augmented by the MACRO-32 compiler register autopreservation feature. Use the **.SET_REGISTERS WRITTEN=<Rn>** directive to augment this set of registers manually.

4. Make sure that each Step 2 driver routine returns control to the operating system with a RET instruction, instead of an RSB instruction.

See *OpenVMS AXP Device Support: Reference* for more information about the Step 2 driver entry point macros.

6.8 Returning Status from Controller and Unit Initialization Routines

An OpenVMS AXP device driver's controller initialization routine and unit initialization routine must return status in R0. If the status returned is not successful, the initialization of your driver is terminated.

6.9 FUNCTAB Macro Changes

An OpenVMS VAX driver contains three or more FUNCTAB macro invocations. For Step 2 drivers, the function decision table (FDT) format is significantly different. Step 2 driver changes include the following:

- The FUNCTAB macro is obsolete.
- The FDT structure consists of a 64-bit mask specifying the buffered functions and a 64-entry vector pointing to the upper-level FDT action routine that corresponds to each of the I/O function codes. There is no bit mask of legal functions.
- Three new macros are used to build the FDT:

  FDT_INI initializes an FDT structure
  FDT_BUF declares the buffered I/O functions
  FDT_ACT declares an upper-level FDT action routine for a set of I/O functions

You must make the following changes:

1. Delete the first FUNCTAB macro, the one that identifies valid I/O function codes, and the FDT label. In their place, insert an FDT_INI macro. The single argument to FDT_INI is the label for the FDT. The label should match the name supplied to the **functb** argument of the DDTAB macro.
2. Replace the second FUNCTAB macro, the one that identifies buffered I/O functions, with an FDT_BUF macro. Replace the word "FUNCTAB" with the word "FDT_BUF" and remove the first null argument.
3. Replace each subsequent FUNCTAB macro with an FDT_ACT macro.

   For example:

```
OpenVMS VAX FDT Declaration

MY_FUNCTBL:

  FUNCTAB ,-      ;legal func
          <SENSEMODE,SENSECHAR,-
           WRITELBLK,WRITEPBLK>

  FUNCTAB ,-      ;buffered func
          <SENSEMODE,SENSECHAR>

  FUNCTAB EXE$SENSE_MODE,-
              <SENSEMODE,SENSECHAR>

  FUNCTAB MY_FDT_WRITE,-
              <WRITELBLK,WRITEPBLK>
```

```
Step 2 FDT Declaration

  FDT_INI MY_FUNCTBL

  FDT_BUF <SENSEMODE,SENSECHAR>

  FDT_ACT EXE_STD$SENSE_MODE,-
              <SENSEMODE,SENSECHAR>

  FDT_ACT MY_FDT_WRITE,-
              <WRITELBLK,WRITEPBLK>
```

Because Step 2 driver support replaces all system-supplied upper-level FDT action routines with new, callable routines, you must also ensure that each FDT_ACT invocation specifies the correct routine name. Generally, the string "_STD" follows the facility ID and precedes the dollar sign ($) in the routine name. For example, replace the following code:

```
FUNCTAB EXE$SETMODE, -
    <SETCHAR,-
     SETMODE>
```

with:

```
FDT_ACT EXE_STD$SETMODE, -
    <SETCHAR,-
     SETMODE>
```

Table 6–1 identifies the new OpenVMS AXP system-supplied upper-level FDT action routines and the OpenVMS VAX routines they replace.

Table 6–1 OpenVMS AXP Upper-Level FDT Action Routines

Obsolete OpenVMS VAX Routine	OpenVMS AXP FDT Action Routine
ACP$ACCESS	ACP_STD$ACCESS
ACP$ACCESSNET	ACP_STD$ACCESSNET
ACP$DEACCESS	ACP_STD$DEACCESS
ACP$MODIFY	ACP_STD$MODIFY
ACP$MOUNT	ACP_STD$MOUNT
ACP$READBLK	ACP_STD$READBLK
ACP$WRITEBLK	ACP_STD$WRITEBLK
New for Step 2	EXE$ILLIOFUNC
EXE$LCLDSKVALID	EXE_STD$LCLDSKVALID
EXE$MODIFY	EXE_STD$MODIFY
EXE$ONEPARM	EXE_STD$ONEPARM
EXE$READ	EXE_STD$READ
EXE$SENSEMODE	EXE_STD$SENSEMODE
EXE$SETCHAR	EXE_STD$SETCHAR
EXE$SETMODE	EXE_STD$SETMODE
EXE$WRITE	EXE_STD$WRITE
EXE$ZEROPARM	EXE_STD$ZEROPARM

(continued on next page)

Table 6–1 (Cont.) OpenVMS AXP Upper-Level FDT Action Routines

Obsolete OpenVMS VAX Routine	OpenVMS AXP FDT Action Routine
MT$CHECK_ACCESS[1]	MT_STD$CHECK_ACCESS

[1]For information about changes in routine behavior, see *OpenVMS AXP Device Support: Reference*.

For more information about the FDT_INI, FDT_BUF, and FDT_ACT macros and the upper-level FDT action, see *OpenVMS AXP Device Support: Reference*.

Warning

Step 2 device drivers support only a single upper-level FDT action routine per I/O function code. For those functions that require processing by more than one upper-level FDT action routine, you should provide a new **composite** FDT function, which sequentially calls each of the required FDT routines as long as the returned status is successful. For more information about composite routines, see Chapter 7.

6.10 FDT Routine Changes

The Step 2 FDT routine changes you need to make depend on the type of FDT routine your driver includes. This section names and describes types of FDT routines, summarizes the differences between OpenVMS VAX and OpenVMS AXP FDT processing, and specifies the required Step 2 FDT routine changes.

An **upper-level FDT action routine** is a routine listed in a driver's function decision table (FDT) as a result of the driver's invocation of the FDT_ACT macro. FDT dispatching code in the $QIO system service calls an upper-level FDT action routine, passing to it the addresses of the I/O request packet (IRP), process control block (PCB), unit control block (UCB), and channel control block (CCB). An upper-level FDT action routine must return SS$_FDT_COMPL status to the $QIO system service. (See *OpenVMS AXP Device Support: Reference* for a full description of the formal interface to an upper-level FDT action routine.)

OpenVMS provides a set of upper-level FDT action routines, but drivers can also define their own driver-specific upper-level FDT action routines. EXE_STD$READ is an example of a Step 2 upper-level FDT action routine.

An **FDT exit routine** is a routine used by an OpenVMS VAX driver to terminate FDT processing and exit from the $QIO system service. For example, EXE$QIODRVPKT is an FDT exit routine. FDT exit routines use the **RET-under-JSB** mechanism to exit from the $QIO system service. The RET under JSB mechanism is the technique of using a RET instruction to return from a JSB interface routine. This RET instruction causes control to return from the most recent CALL interface routine on the current call tree. This technique unwinds any intervening JSB interface routines without returning to their callers and without restoring any register values that were saved by the unwound JSB routines. In a Step 2 driver, FDT exit routines have been replaced by FDT completion routines.

FDT completion routines are the Step 2 replacements for OpenVMS VAX FDT exit routines. Like FDT exit routines, completion routines complete FDT processing by queuing the I/O request to the appropriate next stage of processing. Unlike FDT exit routines, FDT completion routines return back to their callers and do not rely on the RET-under-JSB mechanism. EXE_STD$QIODRKPT is an example of a Step 2 FDT exit routine.

FDT support routines are routines that are called during FDT processing, but they are not upper-level FDT action routines. They have code paths that call FDT completion routines, but they do not complete FDT processing themselves. OpenVMS VAX FDT support routines must use a JSB interface. OpenVMS provides a set of FDT support routines, but drivers can also include their own support routines. EXE_STD$READCHK is an example of a Step 2 FDT support routine.

For OpenVMS VAX drivers:

- Upper-level FDT action routines are invoked via a JSB interface.
- A return from an upper-level FDT action routine via an RSB instruction returns control back to the FDT dispatch loop.
- FDT support routines are all invoked via a JSB interface.
- Exit from OpenVMS VAX FDT processing, and the $QIO system service is via a RET-under-JSB in an FDT exit routine; for example, EXE$ABORTIO, EXE$QIODRVPKT, and so on.
- The $QIO function-dependent parameters are accessible using AP offsets from within any FDT routine. The AP register points directly to the caller's $QIO parameter P1 value.

In contrast, for Step 2 drivers:

- Upper-level FDT action routines are invoked via a new standard call interface.
- Control is returned from an upper-level FDT action routine via a RET instruction, which exits the FDT dispatcher and returns to the $QIO system service.
- Driver-specific FDT support routines may continue to use JSB interfaces, however OpenVMS-provided FDT support routines should be invoked using the new CALL_x macros.
- FDT completion routines are used instead of FDT exit routines. FDT completion routines return back to their callers with the SS$_FDT_COMPL status. All upper-level FDT action routines must return this status back to the $QIO system service.
- The $QIO function-dependent parameters are accessible only from the IRP (offsets IRP$L_QIO_P1, and so on). The $QIO parameters cannot be accessed using AP register offsets in any Step 2 FDT routines.

6.10.1 Upper-Level Routine Entry Point Changes

If the OpenVMS VAX driver you are converting to Step 2 includes a device-specific upper-level FDT action routine, perform the following tasks:

1. Insert the $DRIVER_FDT_ENTRY macro at the entry points of all the upper-level FDT routines that you define in your driver. (See *OpenVMS AXP Device Support: Reference*.) This macro declares the routine's call entry point and ensures, by default, that all nonscratch registers defined by the OpenVMS Calling Standard are preserved. This macro also invokes the $FDTARGDEF macro, thus allowing the FDT routine to access its arguments at their standard locations with respect to the AP.
2. Ensure that the routine does not read R7 to obtain the low-order 6 bits of the $QIO function code parameter, or R8 to obtain the FDT table entry address. It can instead obtain the function code from the IRP and the start of the Step 2 FDT structure from DDT$PS_FDT_2. Note that the Step 2 FDT format differs from the OpenVMS VAX format.
3. Use the default register PRESERVE list on $DRIVER_FDT_ENTRY macro.
4. Remove any definitions of the P1 through P6 offsets that OpenVMS VAX drivers use to access the $QIO function-dependent parameters. For example, remove the following local symbol definitions:

   ```
   P1 = 0
   P2 = 4
   P3 = 8
   P4 = 12
   P5 = 16
   P6 = 20
   ```

   This will help you to find places where you must use the IRP$L_QIO_Pn offsets instead.
5. Access the $QIO function-dependent parameters using the IRP$L_QIO_Pn offsets instead of AP offsets. For example, you must use:

   ```
   MOVL IRP$L_QIO_P1(R3),R0    ;Get caller's buffer address (P1)
   ```

   instead of:

   ```
   MOVL P1(AP),R0
   ```

6.10.2 FDT Exit Routine Changes

Replace the JMP or JSB instructions to OpenVMS VAX FDT exit routines with the Step 2 macros (listed in Table 6–2) that call FDT completion routines. Use the default value for the **do_ret**=**YES** parameter.

For example, replace:

```
JMP G^EXE$ABORTIO
```

with:

```
CALL_ABORTIO
```

Table 6–2 FDT Completion Routines and Macros

Obsolete OpenVMS VAX FDT Exit Routine	Macro	FDT Completion Routine
EXE$ABORTIO	CALL_ABORTIO	EXE_STD$ABORTIO
EXE$ALTQUEPKT	CALL_ALTQUEPKT[1]	EXE_STD$ALTQUEPKT
EXE$FINISHIO	CALL_FINISHIO	EXE_STD$FINISHIO
EXE$FINISHIOC	CALL_FINISHIOC	EXE_STD$FINISHIO
New for Step 2	CALL_FINISHIO_NOIOST	EXE_STD$FINISHIO
EXE$IORSNWAIT	CALL_IORSNWAIT	EXE_STD$IORSNWAIT
EXE$QIOACPPKT	CALL_QIOACPPKT	EXE_STD$QIOACPPKT
EXE$QIODRVPKT	CALL_QIODRVPKT	EXE_STD$QIODRVPKT
EXE$QIORETURN	none	none[2]

[1]The CALL_ALTQUEPKT macro does not provide the **do_ret** argument. An FDT routine that invokes CALL_ALTQUEPKT must typically manage the dispatching of I/O requests to the driver's alternate start-I/O entry point.

[2]If your driver issues a JSB or JMP instruction to EXE$QIORETURN, you must replace the JSB or JMP with code that:

a. Releases the device lock if held. EXE$QIORETURN contained code that unconditionally released the device lock.
b. Places SS$_FDT_COMPL status in R0 before returning to its caller. Because the final system service status in the FDT_CONTEXT structure is SS$_NORMAL by default, your driver need do nothing special to deliver a success status to the $QIO caller.

If you call an FDT completion routine directly (that is, not using a macro), you should note that FDT completion routines:

- Always return to their caller and not to the system service dispatcher.
- Always return the warning status SS$_FDT_COMPL.
- Place the $QIO system service status in a new structure called the FDT_CONTEXT structure.

See *OpenVMS AXP Device Support: Reference* for more information about FDT completion routines and a detailed description of the macros.

6.10.3 OpenVMS-Supplied FDT Support Routine Changes

For Step 2 drivers, replace any JSB instruction to an OpenVMS supplied FDT support routine with the appropriate JSB-replacement macro. (See Table 6–3.) The macros do the following:

- Use the input registers for the corresponding OpenVMS VAX FDT support routine as implicit inputs.
- Call the new Step 2 support routine passing the register values in the correct Step 2 parameter order.
- Restore the output values into the output registers for the corresponding OpenVMS VAX routine.

- Generate code that checks the returned status and invokes a RET instruction on an error. (Some OpenVMS VAX FDT support routines never returned to their callers in the event of an error.)

Table 6–3 System-Supplied FDT Support Routines

Obsolete OpenVMS VAX FDT Support Routine	Macro	FDT Support Routine
EXE$MODIFYLOCK	CALL_MODIFYLOCK	EXE_STD$MODIFYLOCK
EXE$MODIFYLOCK_ERR	CALL_MODIFYLOCK_ERR	EXE_STD$MODIFYLOCK
EXE$READCHK	CALL_READCHK	EXE_STD$READCHK
EXE$READCHKR	CALL_READCHKR	EXE_STD$READCHK
EXE$READLOCK	CALL_READLOCK	EXE_STD$READLOCK
EXE$READLOCK_ERR	CALL_READLOCK_ERR	EXE_STD$READLOCK
COM$SETATTNAST	CALL_SETATTNAST	COM_STD$SETATTNAST
COM$SETCTRLAST	CALL_SETCTRLAST	COM_STD$SETCTRLAST
EXE$WRITECHK	CALL_WRITECHK	EXE_STD$WRITECHK
EXE$WRITECHKR	CALL_WRITECHKR	EXE_STD$WRITECHK
EXE$WRITELOCK	CALL_WRITELOCK	EXE_STD$WRITELOCK
EXE$WRITELOCK_ERR	CALL_WRITELOCK_ERR	EXE_STD$WRITELOCK

See *OpenVMS AXP Device Support: Reference* for further discussion of system-supplied FDT support routines and details about the macros.

6.10.4 Driver-Supplied FDT Support Routine Changes

It is easiest to use your current JSB interfaces for all driver-supplied FDT support routines. In fact, the correct operation of the CALL_x macros depends on keeping the JSB interfaces for your support routines.

To convert an OpenVMS VAX driver that contains driver-supplied FDT support routines to the Step 2 interface, do the following:

1. Use the $DRIVER_FDT_ENTRY macro for upper-level routines with the default preserve list, regardless of the registers that are actually modified by the upper-level FDT routine.
2. Use the FDT completion macros with DO_ RET=YES (the default) and the FDT support routines in Table 6–3.
3. Keep the JSB interface for all driver-supplied FDT support routines.

   This means that you must insert the .JSB_ENTRY directive at the entry points of all the FDT support routines that you define. You must also identify the appropriate register lists for the INPUT, OUTPUT, and SCRATCH parameters on each of your .JSB_ENTRY directives. The correct register lists are determined by the input and output registers that your routine provides. It is crucial that you list the correct OUTPUT registers.

   If you want to convert driver-supplied FDT support routines to CALL interfaces, see Chapter 7. For additional information about the .JSB_ENTRY directive, see *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code*

4. Access the $QIO function-dependent parameters using the IRP$L_QIO_Pn offsets instead of AP offsets. For example, you must use:

```
MOVL IRP$L_QIO_P2(R3),R1      ;Get caller's P2 parameter
```

instead of:

```
MOVL P2(AP),R0
```

6.10.5 Returning from Upper-Level Routines

In most cases, upper-level FDT action routines end with a call to an FDT completion macro that includes a RET instruction. However, if after following the steps outlined in Section 6.10.1 through Section 6.10.4, you still have an RSB instruction in your upper-level FDT action routine, you should change it to the following:

```
MOVL #SS$_NORMAL,R0
RET
```

Encountering an RSB instruction in your upper-level FDT action routine indicates that the upper-level FDT action routine, which you are converting, is one of several upper-level routines called for a single I/O function. Because Step 2 drivers can have only one upper-level FDT action routine for each I/O function, you must also make this FDT routine a composite FDT routine. For information about composite FDT routines, see Section 7.1.

6.11 Adding .JSB_ENTRY Directives

Previous sections of this chapter describe the following topics:

- Guidelines for converting some JSB interface routines to call interfaces
- The required use of the new $DRIVER_xxx_ENTRY entry point macros
- The use of the .JSB_ENTRY directive to identify the entry points of some routines that either can or must retain a JSB interface

After you follow these guidelines, you must identify the entry points of any remaining JSB interface routines in your driver by using the .JSB_ENTRY directive. You must also identify the appropriate register lists for the INPUT, OUTPUT, and SCRATCH parameters on each of your .JSB_ENTRY directives. The correct register lists are determined by the input and output registers that your routine provides. It is crucial that you list the correct OUTPUT registers. For more information about the .JSB_ENTRY directive, see *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code.*

Note

The FORK_ROUTINE macro is a convenient way to declare the entry point of any fork routines that you define.

6.12 Common OpenVMS-Supplied EXEC Routines

Replace any JSB to the routines listed in Table 6–4 with the appropriate macro. If the interface provided by the JSB-replacement macro differs from the original JSB interface, the macro generates a compile-time warning. The compile-time warning identifies the register output that is not provided by the replacement macro. After you have made sure that your code does not depend on this output you can disable the warning by using the INTERFACE_WARNING=NO parameter on the macro.

Certain macros ensure compatibility with the original JSB interface by saving R0, R1, or both. These macros provide an argument that allows you to specify that these registers not be saved. See *OpenVMS AXP Device Support: Reference* for a detailed description of the macros.

Most of the JSB-based routines listed in Table 6–4 continue to be available to Step 2 drivers. However, in many cases, the new call-based interface routine provides better performance than the JSB-based interfaces. If you intend to call a call-based system routine directly (without using a macro), check the "Notes for Converting Step 1 Drivers" section of the routine's description in *OpenVMS AXP Device Support: Reference* to verify the routine interface. You can optimize performance of the macro by following the recommendations listed in Chapter 7.

Table 6–4 Replacement Macros for JSB System Routines

JSB Routine	Replacement Macro	Interface Warning	Save R0/R1
ACP$ACCESS[1]	CALL_ACCESS	No	No
ACP$ACCESSNET[1]	CALL_ACCESSNET	No	No
ACP$DEACCESS[1]	CALL_DEACCESS	No	No
ACP$MODIFY[1]	CALL_ACP_MODIFY	No	No
ACP$MOUNT[1]	CALL_MOUNT	No	No
ACP$READBLK[1]	CALL_READBLK	No	No
ACP$WRITEBLK[1]	CALL_WRITEBLK	No	No
COM$DELATTNAST	CALL_DELATTNAST	No	No
COM$DELATTNASTP	CALL_DELATTNASTP	No	No
COM$DELCTRLAST	CALL_DELCTRLAST	No	No
COM$DELCTRLASTP	CALL_DELCTRLASTP	No	No
COM$DRVDEALMEM	CALL_DRVDEALMEM	No	No
COM$FLUSHATTNS	CALL_FLUSHATTNS	No	No
COM$FLUSHCTRLS	CALL_FLUSHCTRLS	No	No
COM$POST	CALL_POST	No	No
COM$POST_NOCNT	CALL_POST_NOCNT	No	No
COM$SETATTNAST[1]	CALL_SETATTNAST	No	No
COM$SETCTRLAST[1]	CALL_SETCTRLAST	No	No
ERL$ALLOCEMB	CALL_ALLOCEMB	No	No

[1]The JSB-based OpenVMS VAX routine is not supported by the OpenVMS AXP operating system Version 6.1.

(continued on next page)

Table 6–4 (Cont.) Replacement Macros for JSB System Routines

JSB Routine	Replacement Macro	Interface Warning	Save R0/R1
ERL$DEVICEATTN	CALL_DEVICEATTN	No	No
ERL$DEVICERR	CALL_DEVICERR	No	No
ERL$DEVICTMO	CALL_DEVICTMO	No	No
ERL$RELEASEMB	CALL_RELEASEMB	No	No
EXE$ABORTIO[1]	CALL_ABORTIO	No	No
EXE$ALLOCBUF	CALL_ALLOCBUF	No	No
EXE$ALLOCIRP	CALL_ALLOCIRP	No	No
EXE$ALTQUEPKT	CALL_ALTQUEPKT	No	No
EXE$CARRIAGE	CALL_CARRIAGE	No	No
EXE$CHKCREACCES	CALL_CHKCREACCES	No	R1
EXE$CHKDELACCES	CALL_CHKDELACCES	No	R1
EXE$CHKEXEACCES	CALL_CHKEXEACCES	No	R1
EXE$CHKLOGACCES	CALL_CHKLOGACCES	No	R1
EXE$CHKPHYACCES	CALL_CHKPHYACCES	No	R1
EXE$CHKRDACCES	CALL_CHKRDACCES	No	R1
EXE$CHKWRTACCES	CALL_CHKWRTACCES	No	R1
EXE$FINISHIO[1]	CALL_FINISHIO	No	No
EXE$FINISHIOC[1]	CALL_FINISHIOC	No	No
EXE$INSERT_IRP	CALL_INSERT_IRP	No	No
EXE$INSIOQ	CALL_INSIOQ	No	No
EXE$INSIOQC	CALL_INSIOQC	No	No
EXE$IORSNWAIT[1]	CALL_IORSNWAIT	No	No
EXE$LCLDSKVALID[1]	CALL_LCLDSKVALID	No	No
EXE$MNTVERSIO	CALL_MNTVERSIO	No	No
EXE$MODIFY[1]	CALL_EXE_MODIFY	No	No
EXE$MODIFYLOCK[1]	CALL_MODIFYLOCK	No	No
EXE$MODIFYLOCK_ERR[1]	CALL_MODIFYLOCK_ ERR	Yes	No
EXE$MOUNT_VER	CALL_MOUNT_VER	No	R0 and R1
EXE$ONEPARM[1]	CALL_ONEPARM	No	No
EXE$PRIMITIVE_FORK	FORK[2]	No	No
EXE$PRIMITIVE_FORK_WAIT	FORK_WAIT[2]	No	No
EXE$QIOACPPKT[1]	CALL_QIOACPPKT	No	No
EXE$QIODRVPKT[1]	CALL_QIODRVPKT	No	No
EXE$QXQPPKT[1]	CALL_QXQPPKT	No	No
EXE$READCHK [1]	CALL_READCHK	No	No

[1]The JSB-based OpenVMS VAX routine is not supported by the OpenVMS AXP operating system Version 6.1.

[2]The standard call interface version of the routine is used by the macro if the ENVIRONMENT=CALL parameter is specified.

(continued on next page)

Table 6–4 (Cont.) Replacement Macros for JSB System Routines

JSB Routine	Replacement Macro	Interface Warning	Save R0/R1
EXE$READCHKR[1]	CALL_READCHKR	No	No
EXE$READLOCK[1]	CALL_READLOCK	No	No
EXE$READLOCK_ERR[1]	CALL_READLOCK_ ERR	Yes	No
EXE$SENSEMODE[1]	CALL_SENSEMODE	No	No
EXE$SETCHAR[1]	CALL_SETCHAR	No	No
EXE$SETMODE[1]	CALL_SETMODE	No	No
EXE$SNDEVMSG	CALL_SNDEVMSG	No	No
EXE$WRITE[1]	CALL_WRITE	No	No
EXE$WRITECHK[1]	CALL_WRITECHK	No	No
EXE$WRITECHKR[1]	CALL_WRITECHKR	No	No
EXE$WRITELOCK[1]	CALL_WRITELOCK	No	No
EXE$WRITELOCK_ERR[1]	CALL_WRITELOCK_ ERR	Yes	No
EXE$WRTMAILBOX	CALL_WRTMAILBOX	No	No
EXE$ZEROPARM[1]	CALL_ZEROPARM	No	No
IOC$ALTREQCOM	CALL_ALTREQCOM	No	No
IOC$BROADCAST	CALL_BROADCAST	No	R1
IOC$CANCELIO	CALL_CANCELIO	No	R0 and R1
IOC$CLONE_UCB[1]	CALL_CLONE_UCB	Yes	No
IOC$COPY_UCB	CALL_COPY_UCB	No	No
IOC$CREDIT_UCB	CALL_CREDIT_UCB	No	No
IOC$CVTLOGPHY	CALL_CVTLOGPHY	No	No
IOC$CVT_DEVNAM	CALL_CVT_DEVNAM	No	No
IOC$DELETE_UCB	CALL_DELETE_UCB	No	No
IOC$DIAGBUFILL	CALL_DIAGBUFILL	No	No
IOC$FILSPT	CALL_FILSPT	No	No
IOC$GETBYTE	CALL_GETBYTE	No	No
IOC$INITBUFWIND	CALL_INITBUFWIND	No	No
IOC$INITIATE	CALL_INITIATE	No	No
IOC$LINK_UCB[1]	CALL_LINK_UCB	Yes	No
IOC$MAPVBLK	CALL_MAPVBLK	No	No
IOC$MNTVER	CALL_MNTVER	No	No
IOC$MOVFRUSER	CALL_MOVFRUSER	No	No
IOC$MOVFRUSER2	CALL_MOVFRUSER2	No	No
IOC$MOVTOUSER	CALL_MOVTOUSER	No	No
IOC$MOVTOUSER2	CALL_MOVTOUSER2	No	No

[1]The JSB-based OpenVMS VAX routine is not supported by the OpenVMS AXP operating system Version 6.1.

(continued on next page)

Table 6–4 (Cont.) Replacement Macros for JSB System Routines

JSB Routine	Replacement Macro	Interface Warning	Save R0/R1
IOC$PARSDEVNAM	CALL_PARSDEVNAM	No	No
IOC$POST_IRP	CALL_POST_IRP	No	No
IOC$PRIMITIVE_REQCHANH[1]	REQCHAN	No	No
IOC$PRIMITIVE_REQCHANL[1]	REQCHAN	No	No
IOC$PRIMITIVE_WFIKPCH	WFIKPCH	No	No
IOC$PRIMITIVE_WFIRLCH	WFIRLCH	No	No
IOC$PTETOPFN	CALL_PTETOPFN	No	R0 and R1
IOC$QNXTSEG1	CALL_QNXTSEG1	No	No
IOC$RELCHAN	RELCHAN	No	No
IOC$REQCOM	REQCOM	No	No
IOC$SEARCHDEV	CALL_SEARCHDEV	No	No
IOC$SEARCHINT	CALL_SEARCHINT	No	No
IOC$SEVER_UCB	CALL_SEVER_UCB	No	No
IOC$SIMREQCOM	CALL_SIMREQCOM	No	No
IOC$THREADCRB	CALL_THREADCRB	No	R0
MMG$IOLOCK	CALL_IOLOCK	No	No
MMG$UNLOCK	CALL_UNLOCK	No	No
MT$CHECK_ACCESS[1]	CALL_CHECK_ ACCESS	Yes	No
SCH$IOLOCKR	CALL_IOLOCKR	No	R1
SCH$IOLOCKW	CALL_IOLOCKW	No	No
SCH$IOUNLOCK	CALL_IOUNLOCK	No	No

[1]The JSB-based OpenVMS VAX routine is not supported by the OpenVMS AXP operating system Version 6.1.

6.13 New, Changed, and Unsupported OpenVMS Driver Macros

Table 6–5 contains a partial list of the OpenVMS driver macros that have changed for OpenVMS AXP. For a complete list of OpenVMS AXP driver macros and more details about them, see *OpenVMS AXP Device Support: Reference*.

Table 6–5 New, Changed, and Unsupported OpenVMS Driver Macros

Macro	Description	Notes
ADPDISP	Causes a branch to a specified address given the existence of a selected adapter characteristic	Not supported
CLASS_UNIT_INIT	Generates the common code that must be executed by the unit initialization routine of all terminal port drivers	Changed

(continued on next page)

Table 6–5 (Cont.) New, Changed, and Unsupported OpenVMS Driver Macros

Macro	Description	Notes
CPUDISP	Causes a branch to a specified address according to the CPU type of the AXP processor executing the code generated by the macro expansion	Changed
CALL_ABORTIO	Invokes FDT completion routine to abort an I/O request. Replacement for JMP EXE$ABORTIO	New
CALL_ALTQUEPKT	Invokes FDT completion routine to queue an I/O request to the driver's alternate start I/O routine. Replacement for JSB EXE$ALTQUEPKT	New
CALL_FINISHIO	Invokes FDT completion routine to finish an I/O request. Replacement for JMP EXE$FINISHIO	New
CALL_FINISHIOC	Invokes FDT completion routine to finish an I/O request. Replacement for JMP EXE$FINISHIOC	New
CALL_IORNSWAIT	Invokes FDT completion routine to wait for a resource that is required for this I/O request. Replacement for JMP EXE$IORSNWAIT	New
CALL_MODIFYLOCK_ ERR	Check buffer for modify access and lock into memory. An error routine is called on any failure before the I/O request is aborted. Replacement for JSB EXE$MODIFYLOCKR. See also $DRIVER_ERRRTN_ENTRY	New
CALL_QIOACPPKT	Invokes FDT completion routine to queue an I/O request to the XQP or an ACP. Replacement for JMP EXE$QIOACPPKT	New
CALL_QIODRVPKT	Invokes FDT completion routine to queue an I/O request to the driver's start I/O routine. Replacement for JMP EXE$QIODRVPKT	New
CALL_READLOCK_ERR	Check buffer for read access and lock into memory. An error routine is called on any failure before the I/O request is aborted. Replacement for JSB EXE$READLOCKR. See also $DRIVER_ ERRRTN_ENTRY	New
CALL_WRITELOCK_ERR	Check buffer for read access and lock into memory. An error routine is called on any failure before the I/O request is aborted. Replacement for JSB EXE$WRITELOCKR. See also $DRIVER_ERRRTN_ENTRY	New
CRAM_ALLOC	Allocates a controller register access mailbox	New
CRAM_CMD	Calculates the COMMAND, MASK, and RBADR fields for a hardware I/O mailbox according to the requirements of a specific I/O interconnect	New
CRAM_DEALLOC	Deallocates a controller register access mailbox	New
CRAM_IO	Queues the hardware I/O mailbox defined within a controller register access mailbox (CRAM) to the mailbox pointer register (MBPR) and awaits the completion of the mailbox transaction	New
CRAM_QUEUE	Queues the hardware I/O mailbox defined within a controller register access mailbox (CRAM) to the mailbox pointer register (MBPR)	New

(continued on next page)

Table 6–5 (Cont.) New, Changed, and Unsupported OpenVMS Driver Macros

Macro	Description	Notes
CRAM_WAIT	Awaits the completion of a hardware I/O mailbox transaction to a tightly coupled I/O interconnect	New
DDTAB	Generates a driver dispatch table (DDT) labeled *devnam*$DDT	Changed
DEVICELOCK	Achieves synchronized access to a device's database as appropriate to the processing environment	Changed
DPTAB	Generates a driver prologue table (DPT) in a program section called $$$105_PROLOGUE	Changed
DPT_STORE	In the context of a DPTAB macro invocation, generates driver structure initialization and reinitialization routines which the driver loading and reloading procedures call to store values in a table or data structure	Changed
DPT_STORE_ISR	In the context of a DPTAB macro invocation, generates the addresses of the code entry point and procedure descriptor of an interrupt service routine and stores them in the interrupt transfer vector block (VEC)	New
DRIVER_CODE	Declares the program section (psect) that contains driver code	New
DRIVER_DATA	Declares the program section (psect) that contains driver data	New
$DRIVER_ALTSTART_ENTRY	Defines the driver alternate start I/O routine entry point for drivers that use the simple fork mechanism and the CALL-based fork routine environment	New
$DRIVER_CANCEL_ENTRY	Defines the driver cancel routine entry point	New
$DRIVER_CANCEL_SELECTIVE_ENTRY	Defines the driver selective cancel routine entry point	New
$DRIVER_CHANNEL_ASSIGN_ENTRY	Defines the driver channel assign routine entry point	New
$DRIVER_CLONEDUCB_ENTRY	Defines the driver cloned UCB routine entry point	New
$DRIVER_CTRLINIT_ENTRY	Defines the driver controller initialization routine entry point	New
$DRIVER_DELIVER_ENTRY	Defines the driver unit delivery routine entry point	New
$DRIVER_ERRRTN_ENTRY	Defines a driver error routine entry point. Error routines are used in conjunction with the CALL_MODIFYLOCK_ERR, CALL_READLOCK_ERR, and CALL_WRITELOCK_ERR macros	New
$DRIVER_CLONEDUCB_ENTRY	Defines the driver cloned UCB routine entry point	New
$DRIVER_FDT_ENTRY	Defines a driver upper-level FDT routine entry point	New

(continued on next page)

Table 6–5 (Cont.) New, Changed, and Unsupported OpenVMS Driver Macros

Macro	Description	Notes
$DRIVER_MNTVER_ ENTRY	Defines the driver mount verification routine entry point	New
$DRIVER_START_ ENTRY	Defines the driver start I/O routine entry point for drivers that use the simple fork mechanism and the CALL-based fork routine environment	New
$DRIVER_UNITINIT_ ENTRY	Defines the driver unit initialization routine entry point	New
FDT_ACT	Specifies an FDT action routine for set of I/O function codes	New
FDT_BUF	Specifies the buffered functions for a function decision table	New
FDT_INI	Initializes the function decision table	New
FORK	Creates a simple fork process on the local processor	Changed
FORK_ROUTINE	Defines a fork routine entry point	New
FORK_WAIT	Inserts a fork block on the fork-and-wait queue	Changed
FORKLOCK	Achieves synchronized access to a device driver's fork database as appropriate to the processing environment	Changed
FUNCTAB	Builds a function decision table entry in an OpenVMS VAX driver	Replaced by FDT_INI, FDT_BUF, FDT_ACT
INVALIDATE_TB	Allows a single page-table entry (PTE) to be modified while any translation buffer entry that maps it is invalidated, or invalidates the entire translation buffer	Replaced by TBI_ALL, TBI_DATA_64, TBI_ SINGLE, and TBI_ SINGLE_64 macros in OpenVMS AXP systems
IOFORK	Creates a fork process on the local processor for a device driver, disabling timeouts from the associated device	Changed
IFNORD, IFNOWRT, IFRD, IFWRT	Determines the read or write accessibility of a range of memory locations	Changed
KP_ALLOCATE_KPB	Creates a KPB and a kernel process stack, as required by the kernel process services	New
KP_DEALLOCATE_KPB	Deallocates a KPB and its associated kernel process stack	New
KP_END	Terminates the execution of a kernel process	New
KP_RESTART	Resumes the execution of a kernel process	New
KP_REQCOM	Invokes device-independent I/O postprocessing from a kernel process	New
KP_STALL_FORK, KP_ STALL_IOFORK	Stall a kernel process in such a manner that it can be resumed by the fork dispatcher	New
KP_STALL_FORK_WAIT	Stalls a kernel process in such a manner that it can be resumed by the software timer interrupt service routine's examination of the fork-and-wait queue	New
KP_STALL_GENERAL	Stalls the execution of a kernel process	New

(continued on next page)

Table 6–5 (Cont.) New, Changed, and Unsupported OpenVMS Driver Macros

Macro	Description	Notes
KP_STALL_REQCHAN	Stalls a kernel process in such a manner that it can be resumed by the granting of a device controller channel	New
KP_STALL_WFIKPCH, KP_STALL_WFIRLCH	Stalls a kernel process in such a manner that it can be resumed by device interrupt processing	New
KP_START	Starts the execution of a kernel process	New
KP_SWITCH_TO_KP_ STACK	Switches to kernel process context	New
LOADALT	Loads a set of Q22–bus alternate map registers	Not supported
LOADMBA	Loads MASSBUS map registers	Not supported
LOADUBA	Loads a set of UNIBUS map registers or a set of the first 496 Q22–bus map registers	Not supported
LOCK	Achieves synchronized access to a system resource as appropriate to the processing environment	Changed
RELALT	Releases a set of Q22–bus alternate map registers allocated to the driver	Not supported
RELDPR	Releases a UNIBUS adapter data path register allocated to the driver	Not supported
RELMPR	Releases a set of UNIBUS map registers or a set of the first 496 Q22–bus map registers allocated to the driver	Not supported
RELSCHAN	Releases all secondary channels allocated to the driver	Not supported
REQALT	Obtains a set of Q22–bus alternate map registers	Not supported
REQCOM	Invokes device-independent I/O postprocessing to complete an I/O request	Changed
REQCHAN	Obtains a controller's data channel	Not supported
REQDPR	Requests a UNIBUS adapter buffered data path	Not supported
REQMPR	Obtains a set of UNIBUS map registers or a set of the first 496 Q22–bus map registers	Not supported
REQPCHAN	Obtains a controller's data channel	Not supported
REQSCHAN	Obtains a secondary MASSBUS data channel	Not supported
SYSDISP	Causes a branch to a specified address according to the type of AXP system executing the code in the macro expansion	New
TBI_ALL	Invalidates the data and instruction translation buffers in their entirety	New
TBI_DATA_64	Invalidates a single 64-bit virtual address in the data translation buffer	New
TBI_SINGLE	Flushes the cached contents of a single page-table entry (PTE) from the data and instruction translation buffers	New
TBI_SINGLE_64	Invalidates a single 64-bit virtual address in both the data and instruction translation buffers	New

(continued on next page)

Table 6–5 (Cont.) New, Changed, and Unsupported OpenVMS Driver Macros

Macro	Description	Notes
TIMEWAIT	Waits for a specified bit to be cleared or set within a specified length of time	Not supported
TIMEDWAIT	Waits a specified interval of time for an event or condition to occur, optionally executing a series of specified instructions that test for various exit conditions	Changed
WFIKPCH, WFIRLCH	Suspends a driver fork thread and folds its context into a fork block in anticipation of a device interrupt or timeout	Changed

6.14 New, Changed, and Unsupported OpenVMS System Routines

Table 6–6 contains a partial list of the OpenVMS system routines that have changed for OpenVMS AXP. For a complete list of OpenVMS AXP system routines and more details about them, see *OpenVMS AXP Device Support: Reference*.

Table 6–6 New, Changed, and Unsupported OpenVMS System Routines

System Routine	Description	Notes
EXE$BUS_DELAY	Allows a system-specific bus delay within a timed wait	New
EXE$DELAY	Provides a short-term simple delay	New
ERL$DEVICERR, ERL$DEVICTMO, ERL$DEVICEATTN	Allocate an error message buffer and record in it information concerning the error	Changed
EXE$FORK	Creates a fork process on the current processor	Replaced by EXE$PRIMITIVE_FORK and EXE_STD$PRIMITIVE_FORK
EXE$FORK_WAIT	Inserts a fork block on the fork-and-wait queue	Replaced by EXE$PRIMITIVE_FORK_WAIT and EXE_STD$PRIMITIVE_FORK_WAIT
EXE$INSERT_IRP	Inserts an IRP into the specified queue of IRPs according to the base priority of the process that issued the I/O request	New
EXE$INSERTIRP	Inserts an IRP into the specified queue of IRPs according to the base priority of the process that issued the I/O request	Replaced by EXE$INSERT_IRP
EXE$IOFORK	Creates a fork process on the current processor for a device driver, disabling timeouts from the associated device	Replaced by EXE$PRIMITIVE_FORK and EXE_STD$PRIMITIVE_FORK
EXE$KP_ALLOCATE_KPB	Creates a KPB and a kernel process stack, as required by the kernel process services	New

(continued on next page)

Table 6–6 (Cont.) New, Changed, and Unsupported OpenVMS System Routines

System Routine	Description	Notes
EXE$KP_DEALLOCATE_KPB	Deallocates a KPB and its associated kernel process stack	New
EXE$KP_END	Terminates the execution of a kernel process	New
EXE$KP_FORK	Stalls a kernel process in such a manner that it can be resumed by the fork dispatcher	New
EXE$KP_FORK_WAIT	Stalls a kernel process in such a manner that it can be resumed by the software timer interrupt service routine's examination of the fork-and-wait queue	New
EXE$KP_RESTART	Resumes the execution of a kernel process	New
EXE$KP_STALL_GENERAL	Stalls the execution of a kernel process	New
EXE$KP_START	Starts the execution of a kernel process	New
EXE_STD$KP_STARTIO	Sets up and starts a kernel process to be used by a device driver	New
EXE$MODIFYLOCK	Validate and prepare a user buffer for a direct-I/O, DMA read/write operation.	Replaced by EXE_STD$MODIFYLOCK and CALL_MODIFYLOCK macro
EXE$MODIFYLOCKR	Validates and prepares a user buffer for a direct-I/O, DMA modify operation.	Replaced by EXE_STD$MODIFYLOCK and CALL_MODIFYLOCK_ERR macro
EXE$PRIMITIVE_FORK, EXE_STD$PRIMITIVE_FORK	Creates a simple fork process on the current processor	New
EXE$PRIMITIVE_FORK_WAIT, EXE_STD$PRIMITIVE_FORK_WAIT	Inserts a fork block on the fork-and-wait queue	New
EXE$READLOCK	Validate and prepare a user buffer for a direct-I/O, DMA read operation.	Replaced by EXE_STD$READLOCK and CALL_READLOCK macro
EXE$READLOCKR	Validates and prepares a user buffer for a direct-I/O, DMA read operation	Replaced by EXE_STD$READLOCK and CALL_READLOCK_ERR macro
EXE$TIMEDWAIT_COMPLETE	Determines whether the time interval of a timed wait has conclude	New
EXE$TIMEDWAIT_SETUP, EXE$TIMEDWAIT_SETUP_10US	Calculate and return the **end-value** used by EXE$TIMEDWAIT_COMPLETE to determine when a timed wait has completed	New
EXE$WRITELOCK	Validate and prepare a user buffer for a direct-I/O, DMA write operation.	Replaced by EXE_STD$WRITELOCK and CALL_WRITELOCK macro

(continued on next page)

Table 6–6 (Cont.) New, Changed, and Unsupported OpenVMS System Routines

System Routine	Description	Notes
EXE$WRITELOCKR	Validates and prepares a user buffer for a direct-I/O, DMA write operation	Replaced by EXE_STD$WRITELOCK and CALL_WRITELOCK_ERR macro
IOC$ALOALTMAP, IOC$ALOALTMAPN, IOC$ALOALTMAPSP	Allocate a set of Q22–bus alternate map registers	Not supported. See the description of IOC$ALLOC_CNT_RES.
IOC$ALOUBAMAP, IOC$ALOUBAMAPN	Allocate a set of UNIBUS map registers or a set of the first 496 Q22–bus map registers	Not supported. See the description of IOC$ALLOC_CNT_RES.
IOC$ALLOC_CNT_RES	Allocates the requested number of items of a counted resource	New
IOC$ALLOC_CRAB	Allocates and initializes a counted resource allocation block (CRAB)	New
IOC$ALLOC_CRCTX	Allocates and initializes a counted resource context block (CRCTX)	New
IOC$ALLOCATE_CRAM	Allocates a controller register access mailbox	New
IOC$CANCEL_CNT_RES	Cancels a thread that has been stalled waiting for a counted resource	New
IOC$CRAM_CMD	Generates values for the command, mask, and remote I/O interconnect address fields of the hardware I/O mailbox that are specific to the interconnect that is the target of the mailbox operation, inserting these values into the indicated mailbox, buffer, or both	New
IOC$CRAM_IO	Queues the hardware I/O mailbox defined within a controller register access mailbox (CRAM) to the mailbox pointer register (MBPR) and awaits the completion of the mailbox transaction	New
IOC$CRAM_QUEUE	Queues the hardware I/O mailbox defined within a controller register access mailbox (CRAM) to the mailbox pointer register (MBPR)	New
IOC$CRAM_WAIT	Awaits the completion of a hardware I/O mailbox transaction to a tightly coupled I/O interconnect	New
IOC$DEALLOC_CNT_RES	Deallocates the requested number of items of a counted resource	New
IOC$DEALLOC_CRAB	Deallocates a counted resource allocation block (CRAB)	New
IOC$DEALLOC_CRCTX	Deallocates a counted resource context block (CRCTX)	New
IOC$DEALLOCATE_CRAM	Deallocates a controller register access mailbox	New

(continued on next page)

Table 6–6 (Cont.) New, Changed, and Unsupported OpenVMS System Routines

System Routine	Description	Notes
IOC$DIAGBUFILL	Fills a diagnostic buffer if the original $QIO request specified such a buffer	Changed
IOC$KP_REQCHAN	Stalls a kernel process in such a manner that it can be resumed by the granting of a device controller channel	New
IOC$KP_WFIKPCH, IOC$KP_ WFIRLCH	Stall a kernel process in such a manner that it can be resumed by device interrupt processing	New
IOC$LOAD_MAP	Loads a set of adapter-specific map registers	New
IOC$LOADALTMAP	Loads a set of alternate Q22–bus map registers	Not supported; see IOC$LOAD_MAP
IOC$LOADMBAMAP	Loads MASSBUS map registers	Not supported; see IOC$LOAD_MAP
IOC$LOADUBAMAP, IOC$LOADUBAMAPA	Load a set of UNIBUS map registers or a set of the first 496 Q22–bus map registers	Not supported; see IOC$LOAD_MAP
IOC$MAP_IO	Maps I/O bus physical address space into an address region accessible by the processor	New
IOC$NODE_FUNCTION	Performs node-specific functions on behalf of a driver, such as enabling or disabling interrupts from a bus slot	New
IOC_STD$PRIMITIVE_ REQCHANH, IOC_ STD$PRIMITIVE_REQCHANL	Request a controller's data channel and, if unavailable, place process in channel wait queue	New
IOC_STD$PRIMITIVE_ WFIKPCH, IOC_ STD$PRIMITIVE_WFIRLCH	Suspend a driver fork thread and fold its context into a fork block in anticipation of a device interrupt or timeout	New
IOC$READ_IO	Reads a value from a previously mapped location in I/O address space	New
IOC$RELALTMAP	Releases a set of Q22–bus alternate map registers	Not supported; see IOC$DEALLOC_CNT_ RES
IOC$RELDATAP	Releases a UNIBUS adapter's buffered data path.	Not supported
IOC$RELMAPREG	Releases a set of UNIBUS map registers or a set of the first 496 Q22–bus map registers	Not supported; see IOC$DEALLOC_CNT_ RES
IOC$REQALTMAP	Allocates sufficient Q22–bus alternate map registers to accommodate a DMA transfer	Not supported; see IOC$ALLOC_CNT_RES
IOC$REQDATAP, IOC$REQDATAPNW	Request a UNIBUS adapter's buffered data path and, optionally, if no path is available, place process in a data-path wait queue	Not supported

(continued on next page)

Table 6–6 (Cont.) New, Changed, and Unsupported OpenVMS System Routines

System Routine	Description	Notes
IOC$REQMAPREG	Allocates sufficient UNIBUS map registers or a sufficient number of the first 496 Q22–bus map registers to accommodate a DMA transfer	Not supported; see IOC$ALLOC_CNT_RES
IOC$REQPCHANH, IOC$REQPCHANL, IOC$REQSCHANH, IOC$REQSCHANL	Request a controller's primary or secondary data channel and, if unavailable, place process in channel wait queue	Not supported
IOC$WFIKPCH, IOC$WFIRLCH	Suspend a driver fork thread and fold its context into a fork block in anticipation of a device interrupt or timeout	Replaced by IOC_STD$PRIMITIVE_WFIKPCH and IOC_STD$PRIMITIVE_WFIRLCH
IOC$WRITE_IO	Writes a value to a previously mapped location in I/O address space	New
IOC$UNMAP_IO	Unmaps a previously mapped I/O address space	New

6.15 Data Structure Field Changes

Various I/O data structure fields that were byte- and word-size on OpenVMS VAX have been changed to a longword in size on OpenVMS AXP. This change was made because an aligned longword or quadword in memory can be much more efficiently read and written on the AXP architecture than a byte or a word.

If your driver image has undefined data structure offsets (usually discovered at link-time), check the data structure for the same field with a different data type tag. For example, if your OpenVMS VAX driver contained the following references:

```
MOVZWL   IRP$W_BOFF(R3),R0
MOVW     R2,UCB$W_BCNT(R5)
```

you would need to change this to the following:

```
MOVL     IRP$L_BOFF(R3),R0
MOVL     R2,UCB$L_BCNT(R5)
```

It is insufficient to change the name of the data field offset. You must also change the type of instruction used to match the width of the new field. In this example, MOVZWL was changed to MOVL and MOVW was changed to MOVL.

If you cannot find a similarly named field in the same data structure, see Section 7.6 for a list of obsolete data structure cells.

6.16 Incorporating Timed Waits and Delays

Drivers are significant consumers of the TIMEWAIT and TIMEDWAIT macros. Additionally, some drivers implement shorter delays using instruction sequences such as PUSHR, POPR, PUSHR, and POPR. The TIMEDWAIT macro, as described in *OpenVMS AXP Device Support: Reference*, provides a delta time expressed in 10 microsecond units. (The TIMEWAIT macro is not available on OpenVMS AXP systems.)

An OpenVMS driver that requires a delay of less than 10 microseconds, using a special VAX instruction sequence to accomplish it, must use the **nsec** argument of the TIMEDWAIT macro to achieve this delay on OpenVMS AXP.

A driver that must wait a fixed period of time without executing any special instructions during the wait can use the EXE$DELAY system routine. (See *OpenVMS AXP Device Support: Reference* for additional information.)

6.17 Porting Terminal Port Drivers

There are some special requirements for producing a Step 2 OpenVMS AXP terminal port driver, as follows:

- Because an OpenVMS AXP terminal port driver cannot share a single DDT with the OpenVMS AXP terminal class driver, the CLASS_UNIT_INIT macro does not write the address of the class driver's DDT into UCB$L_DDT.
- The terminal port driver must invoke the DDTAB macro specifying the **ctrlinit** and **unitinit** arguments, thus creating its own DDT with entries for its controller initialization routine (DDT$PS_CTRLINIT) and unit initialization routine (DDT$L_UNITINIT). CLASS_UNIT_INIT further initializes the port driver's DDT (the address of which it obtains from UCB$L_DDT) by copying to it from the class driver's DDT the procedure values of the class driver's start-I/O routine, function-decision table, cancel-I/O routine, and alternate start-I/O routine.
- OpenVMS VAX terminal port drivers have depended on the the last instruction in routines such as CLASS_GETNXT to load UCB$B_TT_OUTYPE. Therefore ports could successfully use instruction sequences such as the following

  ```
  JSB     @CLASS_GETNXT(Rx)
  BEQL    no_output
  BLSS    string_output
  .
  .
  .
  ```

  Step 2 OpenVMS AXP terminal port drivers must explicitly check the contents of UCB$B_TT_OUTYPE before a conditional branch, as follows:

  ```
  TSTB    UCB$B_TT_OUTYPE(R5)
  BEQL    no_output
  BLSS    string_output
  ```

- If CLASS_GETNXT returns a –1 to UCB$B_TT_OUTYPE, a Step 2 OpenVMS AXP port driver should obtain the address and size of the output string from UCB$L_TT_OUTADR and UCB$W_TT_OUTLEN respectively. Doing so, rather than relying on this information being passed in registers, enhances portability.

6.18 Initializing Devices with Programmable Interrupt Vectors

The driver loading mechanism, as directed by the System Management utility (SYSMAN) command IO CONNECT connects a hardware device to one or more interrupt vectors. Although most devices connected to VAX systems utilize preassigned vector locations, many devices on AXP systems employ programmable interrupt vectors. It is the driver's responsibility to initialize such a device to use the vector or vectors to which it has been connected.

The driver loading mechanism passes this information to drivers in one of two ways:

- For devices with a single interrupt vector, the cell IDB$L_VECTOR contains the vector offset (into the SCB or the ADP vector table).
- For devices with multiple interrupt vectors, the cell IDB$L_VECTOR contains a pointer to a vector data structure, called a vector list extension (VLE), which contains a list of vectors for the device.

6.19 Floating-Point Instructions Forbidden in Drivers

On OpenVMS AXP systems, usage of the floating-point registers is a per-process attribute and recorded in the data structures that describe process context.

An OpenVMS AXP device driver that executes in interrupt mode on the per-process kernel stack of some random process cannot rely on floating-point usage having been enabled in that process. A floating-point instruction issued in interrupt context would have unpredictable and baleful results.

In addition, a driver FDT routine should not issue floating-point instructions inasmuch as it would alter the current process's context in an unanticipated and adverse manner. A context switch for a process for which floating-point usage is enabled is more expensive than one for a process that does not employ the floating-point registers. If the driver enables floating-point usage within a process, it will appear to be enabled randomly and the process will see random performance.

6.20 Replacing Unsupported Coding Practices

This section describes some of the general VAX MACRO coding constructs that you must change when porting VAX MACRO code to OpenVMS AXP.

6.20.1 Stack Usage

The OpenVMS calling standard defines a stack frame format substantially different from that defined by the VAX calling standard. Therefore, some changes to your code are required.

6.20.1.1 References Outside the Current Stack Frame

By monitoring stack depth throughout a VAX MACRO module, the compiler detects references in a routine to data pushed on the stack by its caller and flags them as errors.

Recommended Change

You must eliminate references in a routine to data pushed on the stack by its caller. Use the OpenVMS kernel process services discussed in Section 3.2.

6.20.1.2 Nonaligned Stack References

At routine calls, the compiler octaword-aligns the stack, if the stack is not already octaword-aligned. Some code, when building structures on the stack, makes unaligned stack references or causes the stack pointer to become unaligned. The compiler flags both of these with information-level messages.

Recommended Change

Provide sufficient padding in data elements or structures pushed onto the stack, or change data structure sizes. Because unaligned stack references also have an impact on VAX performance, you should apply these fixes to code designed to execute on both OpenVMS VAX systems and OpenVMS AXP systems.

6.20.2 Branches from JSB Routines into CALL Routines

The compiler flags, with an information-level message, a call from a JSB routine into a CALL routine, if the .JSB_ENTRY saves registers. The reason such a call is flagged is because the procedure's epilogue code to restore the saved registers will not be executed. If the registers do not have to be restored, no change is necessary.

Recommended Change

The .JSB_ENTRY entry routine is probably trying to execute a RET on behalf of its caller. Change the common code in the .CALL_ENTRY to a .JSB_ENTRY that can be invoked from both routines.

For example, consider the following code:

```
ROUT1:  .CALL_ENTRY
        .
        .
        .
X:
        .
        .
        .
        RET
ROUT2:  .JSB_ENTRY INPUT=<R1,R2>, OUTPUT=<R4>, PRESERVE=<R3>
        .
        .
        .
        BRW     X
        .
        .
        .
        RSB
```

To port such code to OpenVMS AXP, break the .CALL_ENTRY routine into two routines, as follows:

```
ROUT1:  .CALL_ENTRY
        .
        .
        .
        JSB     X
        RET
X:      .JSB_ENTRY INPUT=<R1,R2>, OUTPUT=<R4>, PRESERVE=<R3>
        .
        .
        .
        RSB
ROUT2:  .JSB_ENTRY INPUT=<R1,R2>, OUTPUT=<R4>, PRESERVE=<R3>
        .
        .
        .
        JSB     X
        RET
        .
        .
        .
        RSB
```

6.20.3 Modifying the Return Address

There are several frequently used variations of modifying the return address on the stack, from within a JSB routine, to change the flow of control. All must be recoded.

6.20.3.1 Pushing an Address onto the Stack

The compiler detects any attempt to push an address onto the stack (for instance, PUSHAB label) to cause a subsequent RSB to resume execution at that location and flags this practice as an error. (The next RSB would return to the routine's caller.)

Recommended Change

Remove the PUSH of the address, and add an explicit JSB to the target label before the current routine's RSB. This will result in the same control flow. Declare the target label as a .JSB_ENTRY point.

For example, the compiler flags the following code as requiring a source change:

```
ROUT:   .JSB_ENTRY
        .
        .
        .
        PUSHAB  continue_label
        .
        .
        .
        RSB
```

By adding an explicit JSB instruction, you could change the code as follows. Note that you would place the JSB just before the RSB. In the previous version of the code, it is the RSB instruction that transfers control to *continue_label*, regardless of where the PUSHAB occurs. The PUSHAB is removed in the new version, which follows:

```
ROUT:   .JSB_ENTRY
        .
        .
        .
        JSB     continue_label
        RSB
```

6.20.3.2 Removing the Return Address from the Stack

The compiler detects the removal of a return address from the stack (for instance, TSTL (SP)+) and flags this practice as an error. The removal of a return address in VAX code allows a routine to return to its caller's caller.

Recommended Change

Rewrite the routine such that it returns a status value to its caller that indicates that the caller should return to its caller. Alternatively, the initial caller could pass the address of a "continuation routine," to which the lowest level routine can return by means of a JSB instruction. When the continuation routine uses an RSB instruction to transfer control back to the lowest level routine, the lowest level routine must also RSB.

For example, the compiler would flag the following code as requiring a source change:

```
ROUT1:  .JSB_ENTRY
        .
        .
        .
        JSB     ROUT2
        .
        .
        .
        RSB
ROUT2:  .JSB_ENTRY
        .
        .
        .
        JSB     ROUT3           ; May return directly to rout1
        .
        .
        .
        RSB
ROUT3:  .JSB_ENTRY
        .
        .
        .
        TSTL    (SP)+           ; Discard return address
        RSB                     ; Return to caller's caller
```

You could rewrite the code to return a status value, as follows:

```
ROUT2:  .JSB_ENTRY
        .
        .
        .
        JSB     ROUT3
        BLBS    R0,NO_RET       ; Check ROUT3 status return
        RSB                     ; Return immediately if 0
NO_RET:
        .
        .
        .
        RSB
ROUT3:  .JSB_ENTRY
        .
        .
        .
        CLR     R0              ; Specify immediate return from caller
        RSB                     ; Return to caller's caller
```

6.20.3.3 Modifying the Return Address

The compiler detects any attempt to modify the return address on the stack and flags it as an error.

Recommended Change

Rewrite the code that modifies the return address on the stack to return a status value to its caller instead. The status value causes the caller to either branch to a given location or contains the address of a special .JSB_ENTRY routine the caller should invoke. In the latter case, the caller should RSB immediately after the issuing the JSB to special .JSB_ENTRY routine.

For example, the compiler would flag the following code as requiring a source change:

```
ROUT1:  .JSB_ENTRY
        .
        .
        .
        JSB     ROUT2                  ; Might not return
        .
        .
        .
        RSB
ROUT2:  .JSB_ENTRY
        .
        .
        .
        MOVAB   continue_label, (SP)    ; Change return address
        .
        .
        .
        RSB
```

You could rewrite the code to incorporate a return value as follows:

```
ROUT1:  .JSB_ENTRY
        .
        .
        .
        JSB  ROUT2
        TSTL  R0        ; Check for alternate return
        BEQL  NO_RET    ; Continue normally if 0
        JSB          (R0)      ; Call specified routine
        RSB                    ; and return
NO_RET:
        .
        .
        .
        RSB
ROUT2:  .JSB_ENTRY
        CLRL  R0
        .
        .
        .
        MOVAB  continue_label, R0  ; Specify alternate return
        RSB
```

6.20.3.4 Coroutines

Coroutine calls between two routines are generally implemented as a set of JSB instructions within each routine. Each JSB transfers control to a return address on the stack, removing the return address in the process (for instance, by issuing the instruction (JSB @(SP)+). The compiler detects coroutine calls and flags them as errors.

Recommended Change

You must rewrite the routine that initiates the coroutine linkage to pass an explicit callback routine address to the other routine. The coroutine initiator should then invoke the other routine with a JSB instruction.

For example, consider the following coroutine linkage:

```
ROUT1:  .JSB_ENTRY
        .
        .
        .
        JSB     ROUT2                   ; ROUT2 will call back as a coroutine
        .
        .
        .
        JSB     @(SP)+                  ; Coroutine back to ROUT2
        .
        .
        .
        RSB
ROUT2:  .JSB_ENTRY
        .
        .
        .
        JSB     @(SP)+                  ; Coroutine back to ROUT1
        .
        .
        .
        RSB
```

You could change the routines participating in such a coroutine linkage to exchange explicit callback routine addresses (here, in R6 and R7) as follows:

```
ROUT1:  .JSB_ENTRY
        .
        .
        .
        MOVAB   ROUT1_CALLBACK, R6
        JSB     ROUT2
        RSB
ROUT1_CALLBACK:
        .JSB_ENTRY
        .
        .
        .
        JSB     (R7)                    ; Callback to ROUT2
        .
        .
        .
        RSB
ROUT2:  .JSB_ENTRY
        .
        .
        .
        MOVAB   ROUT2_CALLBACK, R7
        JSB     (R6)                    ; Callback to ROUT1
        RSB
ROUT2_CALLBACK:
        .JSB_ENTRY
        .
        .
        .
        RSB
```

To avoid consuming registers, the callback routine addresses could be pushed onto the stack at routine entry. Then, "JSB @(SP)+" instructions could still be used to perform direct JSBs to the callback routines. In the following example, the callback routine addresses are passed in R0, but pushed immediately at routine entry:

```
ROUT1:  .JSB_ENTRY
        .
        .
        .
        MOVAB   ROUT1_CALLBACK, R0
        JSB     ROUT2
        RSB
ROUT1_CALLBACK:
        .JSB_ENTRY
        PUSHL   R0                  ; Push callback address received in R0
        .
        .
        .
        JSB     @(SP)+              ; Callback to ROUT2
        .
        .
        .
        RSB
ROUT2:  .JSB_ENTRY
        PUSHL   R0                  ; Push callback address received in R0
        .
        .
        .
        MOVAB   ROUT2_CALLBACK, R0
        JSB     @(SP)+              ; Callback to ROUT1
        RSB
ROUT2_CALLBACK:
        .JSB_ENTRY
        .
        .
        .
        RSB
```

6.21 Compiling an OpenVMS AXP Driver

The following is an example of a command procedure used to compile driver MYDRIVER.MAR on an OpenVMS AXP system:

```
$ MACRO/MIGRATION/DEBUG MYDRIVER+ALPHA$LIBRARY:LIB.MLB/LIB
```

6.21.1 Using the /OPTIMIZE=NOREFERENCES Option

By default, the MACRO-32 compiler performs certain optimizations on generated OpenVMS AXP code. These optimizations are fully described in *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code.*

One such optimization (REFERENCES) allows the compiler to recognize that the same data is referenced multiple times and, in certain situations, reduces these references to a single reference. For instance:

```
MOVL   4(R5),R6
MOVL   4(R5),R7
```

generates:

```
LDL     R20,4(R5)
MOV     R20,R6
MOV     R20,R7
```

instead of:

```
LDL     R6,4(R5)
LDL     R7,4(R5)
```

Driver code that reads directly from or writes directly to device registers in local I/O space (or does not use the hardware I/O mechanism described in Chapter 2) may be sensitive to this type of optimization. For such code, Digital recommends that you use the switch /OPTIMIZE=NOREFERENCES during compilation.

7

Handling Complex Conversions Situations

This chapter describes the Step 2 conversion situations that might be too unusual or too complex for the conversion guidelines in Chapter 6.

7.1 Composite FDT Routines

A composite FDT routine is required when a single I/O function code must be processed by more than one upper-level FDT routine. Step 2 FDT dispatching only provides for a single upper-level routine for each I/O function code. When this is not sufficient, the general solution is to write a new upper-level FDT routine that sequentially calls each of the required upper-level FDT routines (checking status on return from each call). Another possible solution is to call the required second upper-level FDT routine at the appropriate point in the first upper-level FDT routine. The need for a composite FDT routine is automatically detected at compile time.

The following example shows an OpenVMS VAX FDT declaration.

```
FUNCTAB MY_FDT_ACPCONTROL,-
                <ACPCONTROL>
FUNCTAB ACP$MODIFY,-
                 <ACPCONTROL,MODIFY>
```

Using the guidelines in Section 6.10, you can obtain the following Step 2 declaration:

```
FDT_ACT MY_FDT_ACPCONTROL,-
                <ACPCONTROL>
FDT_ACT ACP_STD$MODIFY,-
                <ACPCONTROL,MODIFY>
```

However, you will receive the following error message when you attempt to compile the driver:

```
%AMAC-E-GENERROR, generated ERROR: 0 Multiple actions defined for function IO$_ACPCONTROL
```

To correct the source of the error, you must do the following:

1. Write a new upper-level FDT routine. This routine is a composite FDT routine that should call all the upper-level FDT routines listed by the FDT_ ACT macros for the function that has multiple actions. For example, you would write a routine like the following:

```
MY_FDT_ACPCONTROL_COMP:
       $DRIVER_FDT_ENTRY
                                  ;  First FDT routine for IO$_ACPCONTROL
           PUSHL   R6      ;    P4 = CCB
           PUSHL   R5      ;    P3 = UCB
           PUSHL   R4      ;    P2 = PCB
           PUSHL   R3      ;    P1 = IRP
           CALLS   #4,MY_FDT_ACPCONTROL
           BLBC    R0,900$  ;    Quit if done

                                  ;  Second FDT routine for IO$_ACPCONTROL
           CALL_ACP_MODIFY

900$:      RET             ;   Return status
```

2. Examine any of your driver-supplied upper-level FDT routines that you call from a composite FDT routine. With the exception of the last routine called in the composite routine, all the others will have at least one RSB exit path in their OpenVMS VAX version. (See Section 6.10.5.) You must convert this RSB as follows:

   ```
   MOVL #SS$_NORMAL,R0
   RET
   ```

   In an OpenVMS VAX driver, the RSB would have returned control to the FDT dispatching loop, so that the next upper-level FDT routine could be invoked. In a Step 2 driver, you must return a successful status, so that your composite FDT routine continues. Remember that the SS$_FDT_COMPL warning status will be returned by an upper-level FDT routine if FDT processing has completed and should not be continued.

3. Remove the function with multiple actions from all FDT_ACT macros. Then add a new FDT_ACT macro that invokes the new composite FDT routine for the function. In this example, you would write:

   ```
   FDT_ACT MY_FDT_ACPCONTROL_COMP, <ACPCONTROL>

   FDT_ACT ACP_STD$MODIFY, <MODIFY>
   ```

In many cases, a simpler solution is also possible. If you have a function that has multiple actions defined by FDT_ACT macros and the first FDT_ACT macro that references that function does not also include other functions, then you could convert your existing upper-level FDT routine into a composite FDT routine. You can do this by inserting the calls for the remaining upper-level FDT routines at the point where the first upper-level FDT routine would have returned to the OpenVMS VAX FDT dispatcher via an RSB instruction. This is the case in the previous example. Thus, if the OpenVMS VAX version of MY_FDT_ACPCONTROL looks like the following:

```
MY_FDT_ACPCONTROL:
     .JSB_ENTRY
     ...            ;driver-specific processing
     RSB            ;return to FDT dispatcher
```

Then the Step 2 composite version would look like the following:

```
MY_FDT_ACPCONTROL:
   $DRIVER_FDT_ENTRY
   ...            ;driver-specific processing
   CALL_ACP_MODIFY
   RET
```

7.2 Error Routine Callback Changes

If driver FDT processing involves specifying an error callback routine as input to one of the OpenVMS VAX FDT support routines, EXE$READLOCK_ERR, EXE$MODIFYLOCK_ERR, or EXE$WRITELOCK_ERR, do the following:

1. Convert the error callback routine to a standard callable routine by using the following entry-point macro:

   $DRIVER_ERRRTN_ENTRY **[preserve=<>] [,fetch=YES]**

   If the error callback routine alters any nonscratch register as defined by the calling standard, you must add it to the preserve list. You can do this by using the **.SET_REGISTERS** directive or the **preserve** parameter on the $DRIVER_ERRRTN_ENTRY macro. For example, many error routines call EXE$DEANONPAGED or EXE$DEANONPGDSIZ, which destroy the contents of R2. You should specify **.SET_REGISTERS WRITTEN=<R2>**.

2. Replace the RSB used by the error callback routine to return to its caller with a RET instruction.

3. Replace the JSB to EXE$READLOCK_ERR, EXE$MODIFYLOCK_ERR, or EXE$WRITELOCK_ERR with the corresponding JSB-replacement macros: CALL_READLOCK_ERR, CALL_MODIFYLOCK_ERR, or CALL_WRITELOCK_ERR.

   For more information, see *OpenVMS AXP Device Support: Reference*.

7.3 Converting Driver-Supplied FDT Support Routines to Call Interfaces

To convert driver-supplied FDT support routines to call interfaces, follow the procedure described in this section. Note that although this method is more efficient than the one described in Chapter 6, it requires that you make more changes to your source code.

1. Decide what the calling convention is for each of your FDT support routines.

2. Replace .JSB_ENTRY with .CALL_ENTRY at support routine entry points.

3. Within your converted support routines, you must refer to the routine parameters using the appropriate AP offsets. One way to do this is to copy the standard parameters into the registers used by the JSB interface.

4. Make sure that all driver-supplied FDT routines return status in R0.

5. All places that invoke your support routines via a JSB instruction must be changed to invoke the modified support routine via a CALLS instruction after having pushed the actual parameter values.

6. After each of these calls, you must also check the return status. For non-success status values (particularly SS$_FDT_COMPL), you must return to your caller.

   Using .JSB_ENTRY and the FDT completion macros, it is possible to write an FDT support routine that does not return to its caller in the event of an error. Once you convert to standard call interfaces, however, the flow of control always returns to the caller of the support routine.

Note

If any informational messages like the following are displayed, you have probably missed a .JSB_ENTRY FDT support routine or a branch between some other .JSB_ENTRY routine and an FDT support routine.

```
%AMAC-I-RETINJSB, RET in JSB_ENTRY
```

Once you have converted all your FDT support routines to standard call interfaces, you can eliminate many of the registers saves and restores that are generated by the default register preserve list on the $DRIVER_FDT_ENTRY macro. The default preserve list on the $DRIVER_FDT_ENTRY macro saves every nonscratch register to protect against a potential RET-under-JSB inside a .JSB_ENTRY FDT support routine. At the very least, you should be able to reduce the preserve list to **PRESERVE=<R2,R9,R10,R11>** to cover the registers that were allowed to be scratched by OpenVMS VAX upper-level FDT routines. You can reduce this list further, if you know that your FDT routine is not altering these registers, or if you rely on the .SET_REGISTERS directive and the register autopreserve feature of the MACRO-32 compiler,

7.4 Converting the Start I/O Code Path to Call Interfaces

Fork, special kernel AST, system timer expiration, and device interrupt timeout routines that are called by the OpenVMS exec can use either a standard call or the traditional JSB interface described in Chapter 6.

To convert the Start I/O Code Path to call standard interfaces in drivers written in MACRO-32, follow the procedure in Section 7.4.1. For a quick summary of the differences between using ENVIRONMENT=CALL and ENVIRONMENT=JSB, see Section 7.4.2. A detailed description of the Start I/O to REQCOM conversion implications for OpenVMS VAX drivers is available in *OpenVMS AXP Device Support: Reference*.

7.4.1 Start I/O Call Interface Conversion Procedure

To convert the Start I/O Code Path to call standard interfaces in drivers written in MACRO-32, follow these steps:

1. Use the $DRIVER_START_ENTRY and $DRIVER_ALTSTART_ENTRY macros to define the driver's start I/O and appropriate alternate start I/O routines.
2. Use the DDTAB macro keywords

   altstart instead of **jsb_altstart**
   start instead of **jsb_start**

3. Use the ENVIRONMENT=CALL keyword parameter on the FORK, FORK_ROUTINE, FORK_WAIT, IOFORK, REQCOM, REQCHAN, REQPCHAN, WFIKPCH, and WFIRLCH macros.
4. Use the FORK_ROUTINE macro (with ENVIRONMENT=CALL), the .CALL_ENTRY directive, or the .ENTRY directive instead of .JSB_ENTRY to define the entry points for driver fork, channel grant, resume from interrupt, and interrupt timeout routines.
5. Use the RET instruction instead of the RSB instruction to return from all of the previous standard call interface routines.

6. Use the scratch registers as defined by the calling standard. Some of the old JSB interface routines were allowed to scratch registers R2 through R5, which are not in the scratch register set as defined by the calling standard. Also, the calling standard allows R0 and R1 to be scratched by a called routine, while some of the JSB interface routines preserve R0 or R1.

7. Use the following code sequence to invoke the driver interrupt resume routine from the driver interrupt service routine:

```
PUSHL    R5                     ;P3 = UCB from R5
PUSHL    UCB$Q_FR4(R5)          ;P2 = FR4 (32-bits)
PUSHL    UCB$Q_FR3(R5)          ;P1 = FR3 (32-bits)
CALLS    #3,@UCB$L_FPC(R5)      ;call driver routine
```

   as a replacement for:

```
MOVL     UCB$Q_FR3(R5),R3       ;R3 = FR3 (32-bits)
MOVL     UCB$Q_FR4(R5),R4       ;R4 = FR4 (32-bits)
JSB      @UCB$L_FPC(R5)         ;call driver routine
```

   If your driver needs to preserve the full 64-bits of its FR3 or FR4 parameters, then it can use the following code sequence. Note that although the following code appears more complex, it results in code that is just as efficient as that produced by the preceding example.

```
MOVX     UCB$Q_FR3(R5),R16      ;R16 = FR3 (64-bits)
MOVX     UCB$Q_FR4(R5),R17      ;R17 = FR4 (64-bits)
PUSHL    R5                     ;P3 = UCB from R5
PUSHL    R17                    ;P2 = 64-bits of R17
PUSHL    R16                    ;P1 = 64-bits of R16
CALLS    #3,@UCB$L_FPC(R5)      ;call driver routine
```

   For more details about this code sequence, see the description of the FORK ROUTINE interface in *OpenVMS AXP Device Support: Reference*.

   The called routine can obtain 64-bit parameter values by declaring its entry point using the FORK_ROUTINE macro or the WFIKPCH macro.

8. Examine the interroutine branches between the previous routines and other routines in the same modules and change these routines to standard call interfaces.

9. If you encounter any of the following MACRO-32 compiler diagnostic messages, examine the relevant source:

```
%AMAC-E-ILLRSBCAL, illegal RSB in CALL_ENTRY routine

%AMAC-I-BRINTOCAL, branch into CALL_ENTRY routine from
          JSB_ENTRY

%AMAC-I-JSBHOME, arglist use in JSB entry requires homed
          arglist in caller

%AMAC-I-RETINJSB, RET in JSB_ENTRY, with non-scratch
          registers
```

   These messages are likely to result from a .JSB_ENTRY routine that needs to be converted to a standard call entry. Note, however, that in some cases you can receive the last three diagnostic messages under acceptable circumstances. If this happens, you should document the reasons and consider disabling the diagnostic message by bracketing the smallest possible section of relevant code as follows:

```
.DSABL  FLAGGING
.
.
.ENABL  FLAGGING
```

In particular, the use of a RET from a JSB entry routine may be allowable in a Step 2 driver in the context of complex FDT routines. (For more information, see Section 6.10.4.) However, if you change the source code to avoid the need for a RET in a JSB routine, you can improve the performance of the code path. (For more information, see Section 7.3.)

7.4.2 Simple Fork Macro Differences

This section summarizes the differences between using the ENVIRONMENT=CALL and ENVIRONMENT=JSB parameters on the following simple fork macros:

FORK
FORK_ROUTINE
FORK_WAIT
IOFORK
REQCHAN
REQPCHAN
REQCOM
WFIKPCH
WFIRLCH

For more information about the parameters on these macros, see *OpenVMS AXP Device Support: Reference*.

7.4.2.1 Fork Routine End Instruction

Some simple fork macros generate an instruction that ends the current routine and returns control to the routine's caller. In a .JSB_ENTRY routine the appropriate end instruction is an RSB. However, a .CALL_ENTRY routine requires a RET instruction. Table 7–1 lists the simple fork macros whose fork routine end instruction is determined by the ENVIRONMENT parameter.

Table 7–1 Fork Routine End Instruction

Macros	ENVIRONMENT=CALL	ENVIRONMENT=JSB
FORK[1]	RET	RSB
FORK_WAIT[1]	RET	RSB
IOFORK[1]	RET	RSB
REQCHAN	RET	RSB
REQPCHAN	RET	RSB
REQCOM	RET	RSB
WFIKPCH	RET	RSB
WFIRLCH	RET	RSB

[1]If you use the CONTINUE parameter, this macro does not generate a fork routine end instruction.

7.4.2.2 Scratch Registers

Using the ENVIRONMENT=CALL parameter affects the list of scratch registers on some simple fork macros. Table 7–2 summarizes the differences in scratch register usage that are visible to the caller's fork thread. All other implicit register inputs and outputs on the simple fork macros are the same.

Table 7–2 Registers Scratched in Caller's Fork Thread

Macros	ENVIRONMENT=CALL	ENVIRONMENT=JSB
FORK	R0,R1 scratched	R0,R1 preserved
	R3,R4 preserved	R3,R4 sratched
FORK_WAIT	R0,R1 scratched	R0,R1 preserved
IOFORK	R0,R1 scratched	R0,R1 preserved
	R3,R4 preserved	R3,R4 scratched

The following example illustrates how dependence on scratch register usage can be hidden in existing code:

```
MY_UNIT_INIT:
     .JSB_ENTRY  INPUT=<R0,R4,R5>,OUTPUT=<R0>
     ...         ;code that doesn't alter R0
     FORK  ROUTINE=MY_UNIT_INIT_FORK
```

This routine does some work and then queues the routine MY_UNIT_INIT_FORK as a fork routine. A unit initialization routine must return a successful status back to its caller. The preceding sample routine does this as follows:

- R0 is set to SS$_NORMAL before entry into the OpenVMS VAX unit initialization routine.
- The FORK macro with the default ENVIRONMENT=JSB setting does not alter R0.
- The FORK macro generates an RSB instruction.

The Step 2 equivalent of this unit initialization routine uses a standard call interface and must use the ENVIRONMENT=CALL parameter on the FORK macro. However, in doing so, the SS$_NORMAL value held in R0 is destroyed. The following example shows how to avoid this problem:

```
MY_UNIT_INIT:
     $DRIVER_UNITINIT_ENTRY
     ...
     FORK   ROUTINE=MY_UNIT_INIT_FORK,-
            ENVIRONMENT=CALL,-
            CONTINUE=10$
 10$: MOVZWL #SS$_NORMAL,R0
     RET
```

7.4.2.3 Fork Routine Entry Point

Some simple fork macros generate a fork routine entry point. The type of entry point generated depends on which ENVIRONMENT parameter you use. The parameters to a traditional JSB interface fork routine are contained in registers R3, R4, and R5. In contrast, the parameters to a standard call fork routine are passed using the standard argument passing mechanism and are referenced using AP offsets. The following macros generate code that copies the standard arguments into registers R3, R4, and R5; thereby, facilitating the conversion of existing JSB interface fork routines to the standard call interface:

FORK
FORK_ROUTINE
FORK_WAIT
IOFORK
REQCHAN
REQPCHAN
WFIKPCH
WFIRLCH

Table 7–3 summarizes the differences in the fork routine entry points generated by the FORK, FORK_ROUTINE, FORK_WAIT, IO_FORK, REQCHAN, REQPCHAN, WFIKPCH, and WFIRLCH macros as determined by the ENVIRONMENT parameter. Note that the FORK, FORK_WAIT, and IOFORK macros do not generate a fork routine entry point if you use the ROUTINE parameter.

Table 7–3 Fork Routine Entry Points

Entry Point Attributes	ENVIRONMENT=CALL	ENVIRONMENT=JSB
Entry directive	.CALL_ENTRY	.JSB_ENTRY
Parameters	Accessed using AP offsets[1]	R3,R4,R5
Parameter fetch	Parameters copied to R3,R4,R5[2]	None
Allowable scratch registers	R0,R1	R0-R4

[1]The symbolic names for the AP offsets are FORKARG$_FR3, FORKARG$_FR4, and FORKARG$_FKB.

[2]The parameter copy can be disabled on the FORK_ROUTINE macro if the FETCH=NO parameter is specified.

7.5 Device Interrupt Timeouts

Device interrupt timeouts are handled differently for Step 2 drivers. For OpenVMS VAX drivers the UCB$L_FPC cell in the device unit control block (UCB) contained the procedure value of the routine that served as both the resume from interrupt routine and the interrupt timeout routine. These two routines are now separate. The new UCB cell UCB$PS_TOUTROUT is used for the procedure value of the interrupt timeout routine.

These changes are transparent to code that uses the WFIKPCH or WFIRLCH macros, or calls the IOC$PRIMITIVE_WFIKPCH or IOC$PRIMITIVE_WFIRLCH routines. However, code that manually sets the UCB$V_TIM bit in UCB$L_STS now needs to place the timeout routine procedure value into UCB$PS_

TOUTROUT, instead of in UCB$L_FPC. For more information, see the specific routine descriptions in *OpenVMS AXP Device Support: Reference*.

7.6 Obsolete Data Structure Cells

Some DDT and DPT data structure fields that supported OpenVMS VAX device drivers have been removed. Table 7–4 lists the obsolete OpenVMS VAX fields and the OpenVMS AXP fields that have similar functions.

Note that the OpenVMS AXP cells use different names because they point to routines whose interfaces are different or they point to data structures whose layout is significantly altered. For this reason, do not replace each reference to an obsolete OpenVMS VAX field with its corresponding Step 2 field without considering the routine interface and data structure changes.

Table 7–4 Obsolete Data Structure Cells

Obsolete OpenVMS VAX Field	Similar OpenVMS AXP Field
DDT$L_ALTSTART	DDT$PS_ALTSTART_2 or DDT$PS_ALTSTART_JSB
DDT$PS_ALTSTART	DDT$PS_ALTSTART_2 or DDT$PS_ALTSTART_JSB
DDT$L_CANCEL	DDT$PS_CANCEL_2
DDT$PS_CANCEL	DDT$PS_CANCEL_2
DDT$L_CANCEL_SELECTIVE	DDT$PS_CANCEL_SELECTIVE_2
DDT$PS_CANCEL_SELECTIVE	DDT$PS_CANCEL_SELECTIVE_2
DDT$L_CHANNEL_ASSIGN	DDT$PS_CHANNEL_ASSIGN_2
DDT$PS_CHANNEL_ASSIGN	DDT$PS_CHANNEL_ASSIGN_2
DDT$L_CLONEDUCB	DDT$PS_CLONEDUCB_2
DDT$PS_CLONEDUCB	DDT$PS_CLONEDUCB_2
DDT$L_CTRLINIT	DDT$PS_CTRLINIT_2
DDT$PS_CTRLINIT	DDT$PS_CTRLINIT_2
DDT$L_FDT	DDT$PS_FDT_2
DDT$PS_FDT	DDT$PS_FDT_2
DDT$L_MNTVER	DDT$PS_MNTVER_2
DDT$PS_MNTVER	DDT$PS_MNTVER_2
DDT$L_REGDUMP	DDT$PS_REGDUMP_2
DDT$PS_REGDUMP	DDT$PS_REGDUMP_2
DDT$L_START	DDT$PS_START_2 or DDT$PS_START_JSB
DDT$PS_START	DDT$PS_START_2 or DDT$PS_START_JSB
DDT$L_UNITINIT	DDT$PS_UNITINIT_2
DDT$PS_UNITINIT	DDT$PS_UNITINIT_2
DPT$PS_DELIVER	DPT$PS_DELIVER_2

7.7 Optimizing Step 2 Drivers

When you have successfully converted an OpenVMS VAX device driver to a Step 2 device driver, you can optimize the driver's performance by performing the tasks covered in Section 7.7.1 through Section 7.7.4.

7.7.1 Using JSB-Replacement Macros

You can replace a JSB to a system routine in an OpenVMS VAX driver with a macro. The JSB-replacement macro uses the same input registers and modifies the same output registers as the corresponding JSB-based routine. In some cases, you can specify that R0, R1, or both R0 and R1 not be saved if the driver does not need them preserved. (These macros have an argument named **save_r0**, **save_r1**, or, **save_r0r1**.) Eliminating unneeded 64-bit saves of these registers is a performance gain.

As mentioned in Chapter 6, you should use the JSB-replacement macros in Table 6–4 instead of an explicit JSB to the listed JSB-interface system routines. A JSB-replacement macro is provided if the JSB-interface routine is no longer available or if the JSB-interface routine is less efficient than the new standard call version of the routine. The JSB-replacement macros use the register inputs and outputs that your existing OpenVMS VAX code expects. However, these macros directly invoke the new Step 2 standard call interface routines.

7.7.2 Avoid Fetching Unused Parameters

You can adapt a driver's use of the driver entry point macros, so that it more closely resembles the behavior of driver routines.

Each driver entry point macro, by default, initializes the general-purpose registers an OpenVMS VAX driver routine expects as input. At the very least, this practice requires a series of register-to-register loads, plus, by virtue of the default behavior of the MACRO-32 compiler (which automatically preserves any register an entry point modifies), a set of 64-bit register save and restore operations. If the execution code path initiated at a driver entry point does not use one or more of the registers defined as OpenVMS VAX input registers, you might consider specifying **fetch**=**NO** and explicitly loading the registers it does use.

7.7.3 Minimizing Register Preserve Lists

Each driver-entry-point macro, by default, preserves a set of registers across a call. The MACRO-32 compiler, by default, preserves those registers the routine explicitly modifies (but not those implicitly modified by a system routine or driver-specific routine it calls). Here, too, if the execution path initiated at a driver entry point does not use one or more of the registers defined as OpenVMS VAX scratch registers, you might consider removing them from the **preserve** mask. Before doing so, carefully examine the chain of execution that proceeds from the entry point to ensure that some inconspicuous code path does not alter a register you would like to remove from the mask.

For instance, the $DRIVER_FDT_ENTRY macro specifies, by default, that registers R2 through R15 be preserved. For certain FDT entry points, you can specify a much smaller set of registers — **preserve**=**<R2,R9,R10,R11>** is usually sufficient. (These registers are allowed to be scratched by OpenVMS VAX FDT routines.)

You can follow this recommendation only if the FDT processing initiated by the upper-level FDT action routine avoids the situation in which a subroutine call initiated by a JSB instruction is concluded by a RET instruction instead of an RSB. A RET under JSB can occur in FDT processing if the upper-level FDT routine issues a JSB to an FDT support routine that invokes an FDT completion macro (see Table 6–2) without specifying **do_ret**=**NO**. The additional RET instruction generated by a default invocation of the macro would return control back to FDT dispatching code in the $QIO system service, and risks the destruction of register context required by that code.

In some cases you may be able to remove all registers from the preserve list. Note that you can select an empty register preserve list for the driver entry point macros only by specifying **PRESERVE**=**NULL**. In contrast, if you specify **PRESERVE**=**<>**, you will get the default value for the register preserve list and not an empty preserve list.

7.7.4 Branching Between Local Routines

The compiler allows a branch from the body of one routine into the body of another routine in the same module. However, because this results in additional overhead in both routines, the compiler reports an information-level message.

If a CALL routine branches into a code path that executes an RSB, an error message is reported. Such a CALL routine, if not corrected, will fail at run time.

If routines that share a code path have different register declarations, the register restores will be done conditionally. That is, the registers written on the stack at routine entry will be the same for both routines, but whether the register is restored depends on which entry point was invoked.

For example:

```
ROUT1:  .JSB_ENTRY OUTPUT=R3
        MOVL    R1, R3          ; R3 is output, not preserved
        BLSS    LAB1
        RSB
ROUT2:  .JSB_ENTRY              ; R3 is not output, and
        MOVL    #4, R3          ; will be auto-preserved
        JSB     ROUT3           ; no registers destroyed
LAB1:   CLRL    R0
        RSB
```

Note

For both routines, R3 is included in the registers saved on the stack at entry. However, at exit, a mask (also in the stack frame) is tested before restoring R3.

Declaring registers that are destroyed in two routines that share code as **scratch** in one but not the other is more expensive than letting the registers be saved and restored. In this case, you should declare the register R3 as **scratch** in ROUT2 because it was scratched in the OpenVMS VAX version of your driver.

Index

D

E