Creating an OpenVMS Alpha Device Driver from an OpenVMS VAX Device Driver

Order Number: AA–R0Y8A–TE

November 1996

This manual describes how to convert an OpenVMS VAX device driver to run on an OpenVMS Alpha system.

Revision/Update Information: This manual supersedes the *Creating an OpenVMS AXP Step 2 Device Driver from an OpenVMS VAX Device Driver*, Version 6.1.

Software Version: OpenVMS Alpha Version 7.1
OpenVMS VAX Version 7.1

Digital Equipment Corporation
Maynard, Massachusetts

November 1996

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.

Digital conducts its business in a manner that conserves the environment and protects the safety and health of its employees, customers, and the community.

© Digital Equipment Corporation 1996. All rights reserved.

The following are trademarks of Digital Equipment Corporation: Bookreader, DECdirect, DECnet, DECwindows, Digital, HCS, MASSBUS, OpenVMS, OpenVMS Cluster, Q–bus, Q22–bus, TURBOchannel, UNIBUS, VAX, VAX DOCUMENT, VAXcluster, VMS, and the DIGITAL logo.

The following are third-party trademarks:

Futurebus/Plus is a registered trademark of Force Computers GMBH, Federal Republic of Germany.

Internet is a registered trademark of Internet, Inc.

OSF is a registered trademark of the Open Software Foundation, Inc.

Windows NT is a registered trademark of Microsoft Corporation.

All other trademarks and registered trademarks are the property of their respective holders.

ZK6362

The OpenVMS documentation set is available on CD–ROM.

Contents

1 Introduction

2 Accessing Device Interface Registers

3 Suspending Driver Execution

4 Allocating Map Registers and Other Counted Resources

5 Synchronization Requirements for OpenVMS Alpha Device Drivers

6 Conversion Guidelines

7 Handling Complex Conversions Situations

8 Device Driver Entry Points

9 System Routines

10 Data Structures

11 MACRO-32 Driver Macros

Index

Examples

Figures

Tables

Preface

This manual describes how to convert an OpenVMS VAX device driver to an OpenVMS Alpha device driver. It explains how you must change OpenVMS VAX driver code to prepare the driver to be compiled, linked, loaded, and run as an OpenVMS Alpha device driver. This manual identifies 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 Alpha operating system.

Intended Audience

Creating an OpenVMS Alpha 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 Alpha operating system.

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 OpenVMS Alpha device driver 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 Alpha operating system.
- Chapter 3 discusses the suspension mechanisms OpenVMS Alpha 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 Alpha device drivers.
- Chapter 6 contains basic guidelines for converting an OpenVMS VAX device driver to an OpenVMS Alpha device driver.
- Chapter 7 provides tips for converting complex or unusual drivers.
- Chapter 8 provides specific information about how each driver entry point is defined and accessed in an OpenVMS Alpha driver.
- Chapter 9 includes OpenVMS system routines that support OpenVMS Alpha drivers.
- Chapter 10 describes the data structures in the I/O database.

- Chapter 11 documents the OpenVMS macros that have been changed or augmented to support OpenVMS Alpha drivers. It also introduces new macros these drivers may use.

Related Documents

Creating an OpenVMS Alpha Device Driver from an OpenVMS VAX Device Driver focuses on the changes that must be made to an existing OpenVMS VAX device driver to produce an equivalent OpenVMS Alpha device driver.

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

Because *Creating an OpenVMS Alpha Device Driver from an OpenVMS VAX Device Driver* only addresses the porting to OpenVMS Alpha 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 Alpha, should see *Porting VAX MACRO Code to OpenVMS Alpha*.

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

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

For additional information on the Open Systems Software Group (OSSG) products and services, access the Digital OpenVMS World Wide Web site. Use the following URL:

`http://www.openvms.digital.com`

Reader's Comments

Digital welcomes your comments on this manual.

Print or edit the online form SYS$HELP:OPENVMSDOC_COMMENTS.TXT and send us your comments by:

Internet	**openvmsdoc@zko.mts.dec.com**
Fax	603 881-0120, Attention: OSSG Documentation, ZKO3-4/U08
Mail	OSSG Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How To Order Additional Documentation

Use the following table to order additional documentation or information. If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Location	Call	Fax	Write
U.S.A.	DECdirect 800−DIGITAL 800−344−4825	Fax: 800−234−2298	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061
Puerto Rico	809−781−0505	Fax: 809−749−8300	Digital Equipment Caribbean, Inc. 3 Digital Plaza, 1st Street, Suite 200 P.O. Box 11038 Metro Office Park San Juan, Puerto Rico 00910−2138
Canada	800−267−6215	Fax: 613−592−1946	Digital Equipment of Canada, Ltd. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: DECdirect Sales
International	—	—	Local Digital subsidiary or approved distributor
Internal Orders	DTN: 264−4446 603−884−4446	Fax: 603−884−3960	U.S. Software Supply Business Digital Equipment Corporation 8 Cotton Road Nashua, NH 03063−1260

ZK−7654A−GE

Conventions

The name of the OpenVMS AXP operating system has been changed to the OpenVMS Alpha operating system. Any references to OpenVMS AXP or AXP are synonymous with OpenVMS Alpha or Alpha.

In this manual, every use of DECwindows and DECwindows Motif refers to DECwindows Motif for OpenVMS software.

The following conventions are also 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* or GOLD *x*	A sequence such as PF1 *x* or GOLD *x* indicates that you must first press and release the key labeled PF1 or GOLD and then press and release another key or a pointing device button. GOLD key sequences can also have a slash (/), dash (–), or underscore (_) as a delimiter in EVE commands.
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.)

. . .	Horizontal ellipsis points in examples indicate 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.
. . .	Vertical ellipsis points indicate the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In command 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 command format descriptions, braces indicate a required choice of options; you must choose one of the options listed.
text style	This text style represents the introduction of a new term or the name of an argument, an attribute, or a reason. This style is also used to show user input in Bookreader versions of the manual.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *device-name* contains up to five alphanumeric characters).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace type`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1

Introduction

OpenVMS Alpha Version 6.1 introduced formal support for user-written device drivers and a new device driver interface known as the **Step 2** driver interface. If you have an existing OpenVMS VAX device driver that you want to run on an Alpha system, and you have not made the changes required for OpenVMS Alpha Version 6.1, you must make the driver interface changes described in this manual.

Note

For OpenVMS Alpha releases prior to OpenVMS Alpha Version 7.1, OpenVMS Alpha device drivers were referred to as Step 2 drivers. For OpenVMS Alpha Version 7.1—unless Step 2 is explicitly required in driver code—references to these drivers are synonymous with OpenVMS Alpha drivers.

1.1 Overview of OpenVMS Alpha Driver Changes

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

- You must identify OpenVMS Alpha 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 Alpha device driver must use multiprocessing synchronization mechanisms, regardless of whether it will operate in an OpenVMS Alpha multiprocessing environment. See Section 5.1.
- An OpenVMS Alpha 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 Alpha 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 Alpha device driver, you must compile the source module or modules with the MACRO-32 compiler for OpenVMS Alpha. 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 Alpha 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.
- Data structures have been greatly overhauled. Fields have been deleted, expanded, and added. Many field aliases have been removed. If your driver uses fields that have been removed from the unit control block (UCB) for OpenVMS Alpha, Digital recommends using the $DEFINI, $DEF, $DEFEND, and associated macros to create the needed fields in a UCB extension.
- OpenVMS Alpha 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.
- 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 Alpha 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 Alpha 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 Alpha Driver Similarities

OpenVMS Alpha 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 Alpha Driver Routine Naming Conventions

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

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 Alpha 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 chapter.

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 the macros chapter.

1.5 Writing OpenVMS Alpha Drivers in C

OpenVMS Alpha provides the support necessary to write a device driver in the C programming language. For information about writing OpenVMS Alpha device drivers in C or another high-level language, see the *Writing OpenVMS Alpha Device Drivers in C* manual.

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

The OpenVMS Alpha driver interface has increased the differences between OpenVMS Alpha and OpenVMS VAX device drivers. A key difference is that while OpenVMS Alpha 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 Alpha 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 Alpha, 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 Alpha 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.

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 Alpha 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 Alpha systems, there are no rules governing how hardware implementations allow access to I/O space. Some Alpha 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 Alpha 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 Alpha 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, Alpha 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 Alpha 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 Alpha operating system is to provide standard ways to access I/O device registers. OpenVMS Alpha 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 Alpha systems, you must map the registers into the processor's virtual address space. OpenVMS Alpha 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 Alpha 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.

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 Alpha 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 Alpha 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 the appropriate chapter in this manual.

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 Alpha 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 Alpha 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 Alpha 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 Alpha 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 Alpha 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 Alpha 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 Alpha 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 Alpha, like other Alpha compilers, cannot allow such absolute control over the stack. A typical routine written in VAX MACRO, and compiled for execution on an OpenVMS Alpha 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 Alpha systems.

Consequently, in creating an OpenVMS Alpha 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 Alpha 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 Alpha 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 Alpha system-specific suspension routines.

OpenVMS Alpha 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 Alpha 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 Alpha Driver Execution

OpenVMS VAX Macro	OpenVMS Alpha 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 Alpha Driver Execution

OpenVMS VAX Macro	OpenVMS Alpha 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 Alpha driver uses to suspend execution.

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

OpenVMS VAX Routine	OpenVMS Alpha 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 Alpha counterpart to the OpenVMS VAX system routines EXE$FORK and EXE$IOFORK. EXE_STD$PRIMITIVE_FORK_WAIT is the OpenVMS Alpha counterpart to the OpenVMS VAX EXE$FORK_WAIT routine.

Use of the simple fork process mechanism in an OpenVMS Alpha 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 the system routines chapter.

The OpenVMS Alpha 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 Alpha routines for most device drivers. The following sections provide some examples of how an OpenVMS Alpha device driver may use these 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 Alpha 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 Alpha system:

```
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
```

---

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

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 Alpha 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 *Porting VAX MACRO Code to OpenVMS Alpha*.

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 Alpha 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 Alpha 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 Alpha 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 Alpha 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 Alpha 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 Alpha counterparts to the OpenVMS VAX system routines IOC$REQPCHANH and IOC$REQPCHANL.

Use of the simple fork process mechanism in an OpenVMS Alpha 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 Alpha 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 the system routines chapter.

The OpenVMS Alpha 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 Alpha 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 Alpha 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 Alpha.

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 the macros chapter 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 Alpha 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 the system routines chapter.

The OpenVMS Alpha 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 Alpha 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$TIMEOUT calls a driver's timeout handling routine directly by means of a VAX MACRO JSB instruction. On an OpenVMS Alpha system, EXE$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 Alpha 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 Alpha 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 Alpha 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 Alpha 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 the macros chapter 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 Alpha 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:

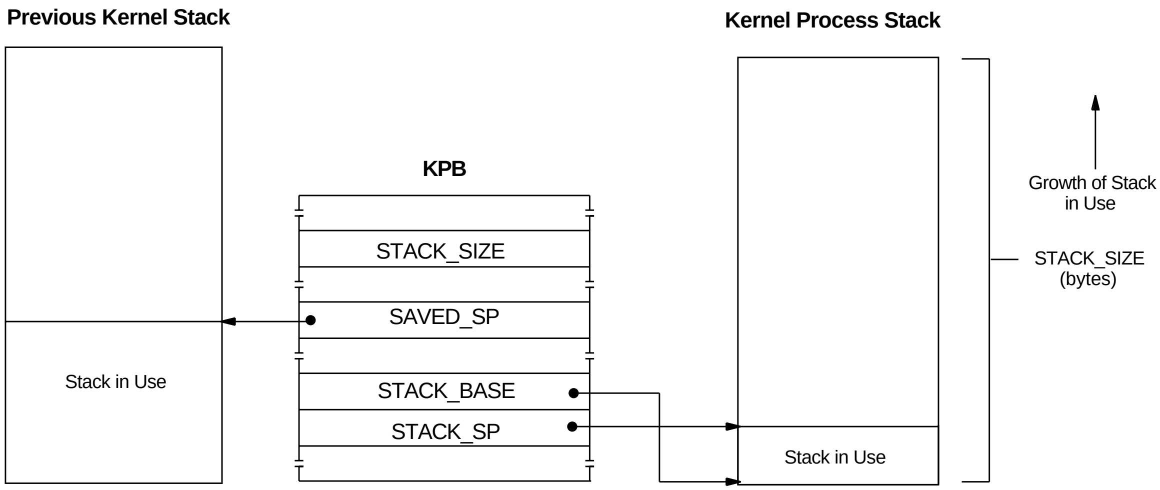
- At initiation, a switch from the current kernel stack to that of the kernel process
- At a stall, a switch from the kernel process private stack to the one current when the kernel process was entered
- At restart, a switch from the current kernel stack to that of the kernel process
- At termination, 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, macro or system routines chapter.

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$KP_STARTIO customizes the kernel process environment specifically for driver kernel processes, facilitating the conversion of OpenVMS VAX drivers that use the simple fork process mechanism to OpenVMS Alpha drivers. To this end, EXE_STD$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 the macro chapter.

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.

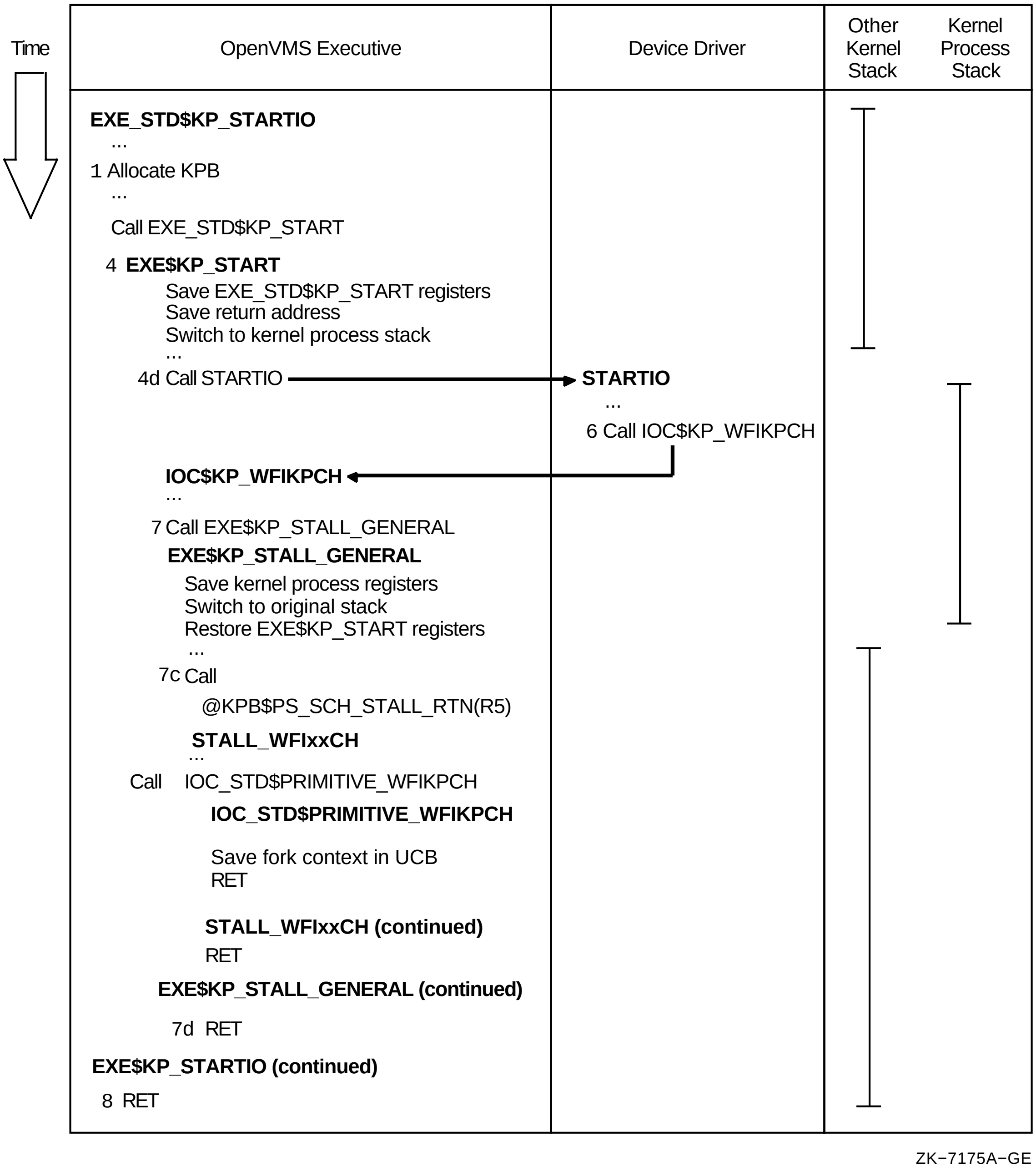
In Figures 3–2, 3–3, and 3–4, two barred lines appear in the rightmost column. Each represents the current stack of execution: either the kernel process private stack or a kernel stack.

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).

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. 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:

    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).

        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.

1. When the device interrupts, Alpha Alpha 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)
```

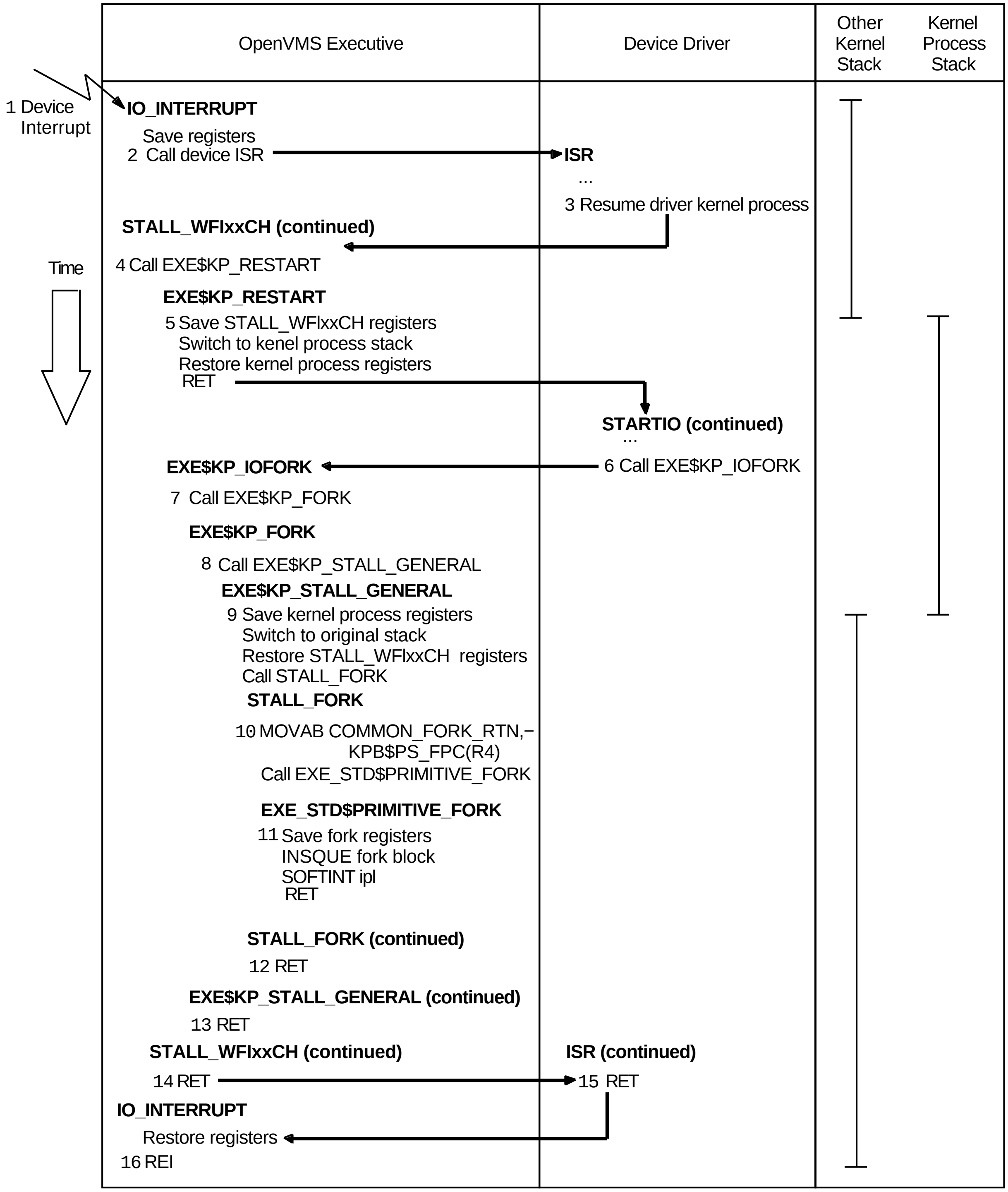
4. STALL_WFIXXCH calls EXE$KP_RESTART.

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).

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_

Figure 3–3 Device Interrupt Resumes Driver Kernel Process



ZK−7176A−GE

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.

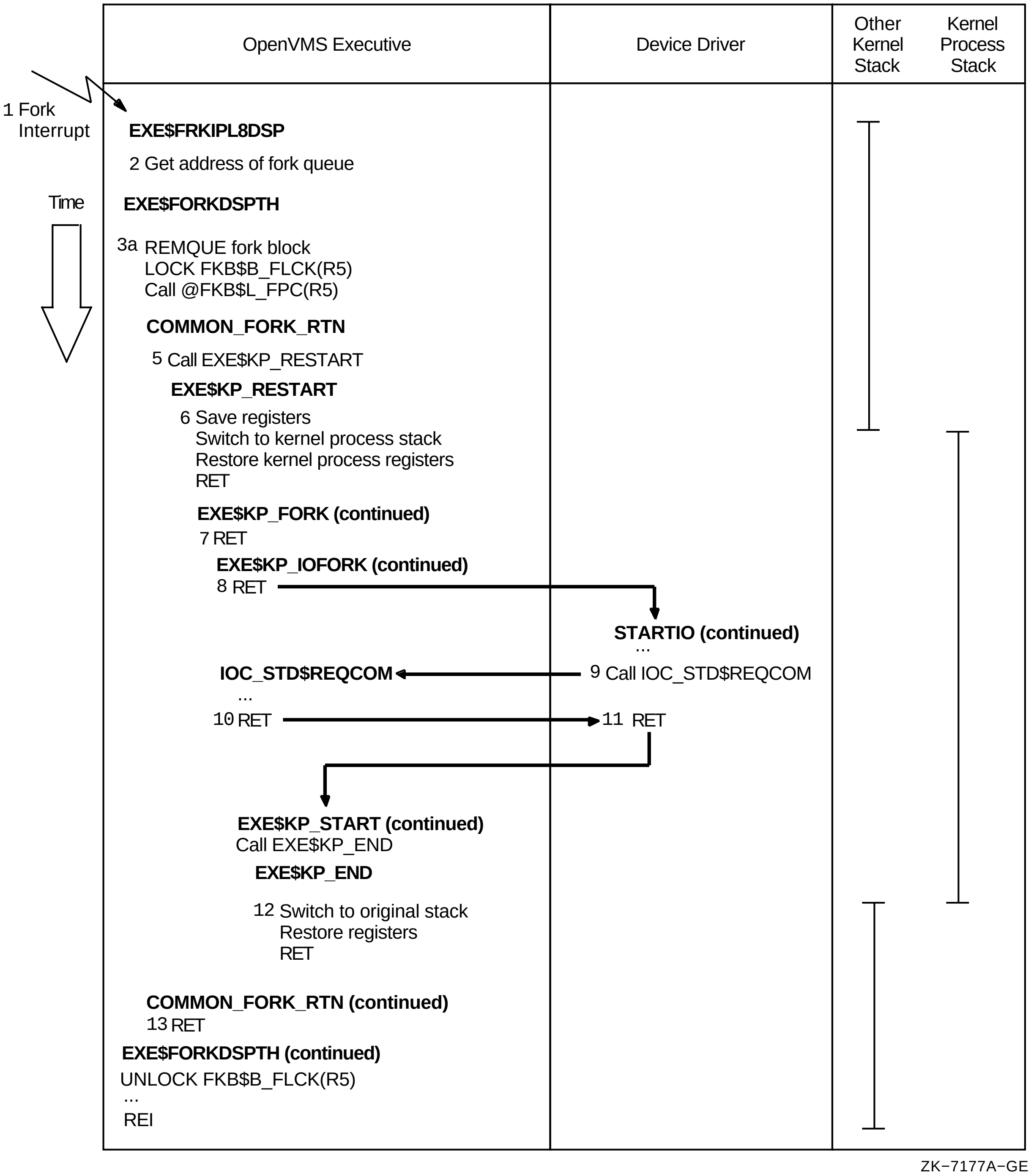
3.2.7.3 Resumption of a Driver Kernel Process by a Fork Interrupt

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.

1. When processor IPL drops below the fork IPL, the fork IPL software interrupt is granted. The fork dispatcher interrupt service routine, EXE$FRKIPL*x*DSP [where *x* is 6, 8, 9, 10, or 11, one of the fork IPLs] is entered. This example assumes a fork IPL of 8.
2. EXE$FRKIPL8DSP obtains the offset to the IPL 8 fork queue listhead and enters EXE$FORKDSPTH.

Figure 3–4 Fork Interrupt Resumes Driver Kernel Process



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 the system routines chapter, 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 Alpha systems do not support the UNIBUS, Q22–bus, and MASSBUS adapters, the OpenVMS Alpha 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 Alpha I/O subsystems that provide map registers, such as the TURBOchannel I/O processor for DEC 3000 Alpha Model 500 systems, OpenVMS Alpha 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 Alpha 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 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.

- 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.

Despite the new structures and new routines, an OpenVMS Alpha 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 Alpha 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.

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$L_ITEM_CNT	Number of items to be allocated. When requesting map registers, this value in this field should include two extra map registers to be allocated and loaded as a guard page to prevent runaway transfers. There may be additional bus-specific requirements.
CRCTX$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 Alpha 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.

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
```

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 Alpha 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
```

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
```

5

Synchronization Requirements for OpenVMS Alpha Device Drivers

This chapter discusses special synchronization requirements for OpenVMS Alpha device drivers beyond the basic synchronization requirements for OpenVMS Alpha device drivers. 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 Alpha 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 Alpha 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 Alpha system.

5.1 Producing a Multiprocessing-Ready Driver

All OpenVMS Alpha device drivers must adhere to the rules for OpenVMS multiprocessing device drivers.

The following is a general summary of those rules for OpenVMS Alpha 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 Alpha 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 Alpha 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 Alpha 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 Alpha 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 Alpha 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 Alpha 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 Data Items

The VAX architecture supports instructions that can read or write byte- and word- sized data in a single noninterruptible operation. The Alpha Alpha architecture supports instructions that read or write longword- and quadword-sized data uninterruptedly. Because the Alpha 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 Alpha 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 Alpha 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 Alpha 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 Alpha 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 *Porting VAX MACRO Code to OpenVMS Alpha* 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 Alpha 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 Alpha 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 Alpha device driver. For more details about the macros, system routines, and entry points listed in this chapter, see the appropriate chapter in this manual. For more details about porting VAX MACRO code to OpenVMS Alpha, see *Porting VAX MACRO Code to OpenVMS Alpha*.

6.1 OpenVMS Alpha Device Driver Program Sections

An OpenVMS Alpha 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 Alpha 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 Alpha driver. Moreover, because OpenVMS Alpha 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 Alpha.

  Unlike its behavior in OpenVMS VAX device drivers, the DDTAB macro does not define the $$$115_DRIVER psect for OpenVMS Alpha device drivers. Rather it defines the data psect ($110_DATA) in which the DDT resides.
- OpenVMS macros that construct data, such as DDTAB and FUNCTAB, automatically invoke the DRIVER_DATA macro prior to creating the data. By default, the DRIVER_DATA macro declares the psect $$$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 Alpha.

- 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 Alpha 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 Alpha uses the value in DPT$IW_STEP to detect driver sources that have not been modified to conform to the currently supported OpenVMS Alpha driver implementation. OpenVMS Alpha uses the value in DPT$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 Alpha driver, the DPT can be in any read/write image section of the driver.

See the driver macros chapter 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 Alpha driver.

6.3.1 DDTAB Routine Name Changes

The routines pointed to by the driver dispatch table (DDT) must conform to OpenVMS Alpha 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 macros chapter 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 Alpha 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,-
                .
                .
                .
```

6.3.3 Simple Fork Mechanism—JSB-Based Fork Routines

Chapter 3 describes alternatives available to OpenVMS Alpha 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 Alpha 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
```

6.5 Interrupt Service Routine Entry Points

The interrupt service routine in an OpenVMS Alpha 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 Alpha 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 Alpha 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 OpenVMS Alpha driver-supplied routines, perform the following tasks:

1. Use the appropriate macro to identify entry points in your driver. OpenVMS Alpha 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 OpenVMS Alpha driver routine returns control to the operating system with a RET instruction, instead of an RSB instruction.

6.8 Returning Status from Controller and Unit Initialization Routines

An OpenVMS Alpha 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 OpenVMS Alpha drivers, the function decision table (FDT) format is significantly different. OpenVMS Alpha 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 OpenVMS Alpha 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 Alpha system-supplied upper-level FDT action routines and the OpenVMS VAX routines they replace.

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

Obsolete OpenVMS VAX Routine	OpenVMS Alpha 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 OpenVMS Alpha	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 Alpha Upper-Level FDT Action Routines

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

[1] For information about changes in routine behavior, see system routines chapter.

Warning

OpenVMS Alpha 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 OpenVMS Alpha 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 Alpha FDT processing, and specifies the required OpenVMS Alpha 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.

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 OpenVMS Alpha 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 an OpenVMS Alpha driver, FDT exit routines have been replaced by FDT completion routines.

FDT completion routines are the OpenVMS Alpha 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 an OpenVMS Alpha 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 an OpenVMS Alpha 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 OpenVMS Alpha 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 OpenVMS Alpha FDT routines.

6.10.1 Upper-Level Routine Entry Point Changes

If the OpenVMS VAX driver you are converting to OpenVMS Alpha 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. 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 $QIO function code parameter, or R8 to obtain the FDT table entry address. It can instead obtain the function code from the IRP and the start of the OpenVMS Alpha FDT structure from DDT$PS_FDT_2. Note that the OpenVMS Alpha 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 OpenVMS Alpha 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 OpenVMS Alpha	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.

6.10.3 OpenVMS-Supplied FDT Support Routine Changes

For OpenVMS Alpha 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 OpenVMS Alpha support routine passing the register values in the correct OpenVMS Alpha 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

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 OpenVMS Alpha 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 *Porting VAX MACRO Code to OpenVMS Alpha*

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 OpenVMS Alpha 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 *Porting VAX MACRO Code to OpenVMS Alpha*.

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.

Most of the JSB-based routines listed in Table 6–4 continue to be available to OpenVMS Alpha 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 the system routines chapter 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$ALLOCEMB	CALL_ALLOCEMB	No	No
ERL$DEVICEATTN	CALL_DEVICEATTN	No	No

---

[1] The JSB-based OpenVMS VAX routine is not supported by the OpenVMS Alpha 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$DEVICERR	CALL_DEVICERR	No	No
ERL$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
EXE$READCHKR[1]	CALL_READCHKR	No	No

---

[1] The JSB-based OpenVMS VAX routine is not supported by the OpenVMS Alpha 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$READLOCK[1]	CALL_READLOCK	No	No
EXE$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
IOC$PARSDEVNAM	CALL_PARSDEVNAM	No	No

---

[1] The JSB-based OpenVMS VAX routine is not supported by the OpenVMS Alpha 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$POST_IRP	CALL_POST_IRP	No	No
IOC$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 Alpha 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 Alpha.

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
CPUDISP	Causes a branch to a specified address according to the CPU type of the Alpha 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

(continued on next page)

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

Macro	Description	Notes
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 $DRIVER_ERRRTN_ENTRY	New
CRAM_ALLOC	Allocates a controller register access mailbox	New
CRAM_CMD	Calculates the COMMAND, MASK, and RBADR fields for a hardware I/O mailbox according to the requirements of a specific I/O interconnect	New
CRAM_DEALLOC	Deallocates a controller register access mailbox	New
CRAM_IO	Queues the hardware I/O mailbox defined within a controller register access mailbox (CRAM) to the mailbox pointer register (MBPR) and awaits the completion of the mailbox transaction	New
CRAM_QUEUE	Queues the hardware I/O mailbox defined within a controller register access mailbox (CRAM) to the mailbox pointer register (MBPR)	New
CRAM_WAIT	Awaits the completion of a hardware I/O mailbox transaction to a tightly coupled I/O interconnect	New
DDTAB	Generates a driver dispatch table (DDT) labeled *devnam*$DDT	Changed

(continued on next page)

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

Macro	Description	Notes
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
$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

(continued on next page)

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

Macro	Description	Notes
$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 Alpha 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
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

(continued on next page)

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

Macro	Description	Notes
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 Alpha 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
TIMEWAIT	Waits for a specified bit to be cleared or set within a specified length of time	Not supported

(continued on next page)

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

Macro	Description	Notes
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 Alpha.

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
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

(continued on next page)

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

System Routine	Description	Notes
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
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

(continued on next page)

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

System Routine	Description	Notes
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
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

(continued on next page)

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

System Routine	Description	Notes
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
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

(continued on next page)

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

System Routine	Description	Notes
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 Alpha. This change was made because an aligned longword or quadword in memory can be much more efficiently read and written on the Alpha 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 provides a delta time expressed in 10 microsecond units. (The TIMEWAIT macro is not available on OpenVMS Alpha 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 Alpha.

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.

6.17 Porting Terminal Port Drivers

There are some special requirements for producing an OpenVMS Alpha terminal port driver, as follows:

- Because an OpenVMS Alpha terminal port driver cannot share a single DDT with the OpenVMS Alpha 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
  .
  .
  .
  ```

  OpenVMS Alpha 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$B_TT_OUTYPE, an OpenVMS Alpha port driver should obtain the address and size of the output string from UCB$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 Alpha 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 Alpha systems, usage of the floating-point registers is a per-process attribute and recorded in the data structures that describe process context.

An OpenVMS Alpha 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 Alpha.

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 Alpha 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 Alpha, 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 Alpha Driver

The following is an example of a command procedure used to compile driver MYDRIVER.MAR on an OpenVMS Alpha 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 Alpha code. These optimizations are fully described in *Porting VAX MACRO Code to OpenVMS Alpha*.

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 OpenVMS Alpha 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. OpenVMS Alpha 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 OpenVMS Alpha 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 an OpenVMS Alpha 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 OpenVMS Alpha 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.

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.

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 the system routines chapter.

   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 an OpenVMS Alpha 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

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 R3,R4 preserved	R0,R1 preserved R3,R4 sratched
FORK_WAIT	R0,R1 scratched	R0,R1 preserved
IOFORK	R0,R1 scratched R3,R4 preserved	R0,R1 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 OpenVMS Alpha 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 OpenVMS Alpha 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.

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 Alpha fields that have similar functions.

Note that the OpenVMS Alpha 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 OpenVMS Alpha field without considering the routine interface and data structure changes.

Table 7–4 Obsolete Data Structure Cells

Obsolete OpenVMS VAX Field	Similar OpenVMS Alpha 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 OpenVMS Alpha Drivers

When you have successfully converted an OpenVMS VAX device driver to an OpenVMS Alpha 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 OpenVMS Alpha 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.

8

Device Driver Entry Points

This chapter describes the standard driver routines that OpenVMS Alpha uses as entry points in a device driver program.

Unlike OpenVMS VAX, OpenVMS Alpha does not support driver unloading routines and unsolicited interrupt handling routines.

This chapter also describes the Alpha driver-entry-point macros that replace the .JSB_ENTRY directive used in OpenVMS VAX driver entry points. These macros perform the following operations:

1. Declare a call entry point.
2. Specify a register save list that consists of the registers that the Step 1 JSB interface was allowed to scratch. This save list augments the list of autopreserved registers detected by the MACRO-32 compiler. You can specify an alternative save list if you are certain that the default mask contains registers that are not used in the execution path initiated by the entry point.
3. Define symbolic AP offsets that correspond to the routine parameters.
4. Copy the input parameters into the registers that correspond to the input registers of the JSB interface. You can disable this register loading by using an optional parameter.

Alternate Start-I/O Routine

Initiates activity on a device that can support multiple, concurrent I/O operations and synchronizes access to its UCB.

Format

ALTSTART (irp, ucb)

Arguments

Argument	Type	Access	Mechanism
irp	IRP	input	reference
ucb	UCB	input	reference

irp

I/O request packet for the current I/O request

ucb

Unit control block of the device that is the target of the I/O request

Essentials

Identifying the Routine

Specify the address of the alternate start-I/O routine in the **altstart** argument to the DDTAB macro. This macro places the procedure value of the routine into the DDT.

Declaring the Entry Point

Use:

`$DRIVER_ALTSTART_ENTRY [preserve=<R2,R3,R4,R5>] [,fetch=YES]`

where:

> **preserve** indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the alternate start-I/O routine
> **fetch**=**YES**, the default, loads the addresses of the IRP and UCB into R3 and R5, respectively; **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver alternate start-I/O routine that uses this macro can access any of its arguments by using a symbolic name of the form ALTARG$**_argument-name**.

Called by

Called by routine EXE_STD$ALTQUEPKT in module SYSQIOREQ. A driver FDT routine typically is the caller of EXE_STD$ALTQUEPKT.

Context

An alternate start-I/O routine begins execution at fork IPL, holding the corresponding fork lock. It must return control to EXE_STD$ALTQUEPKT in this context.

Because an alternate start-I/O routine gains control in fork process context, it can access only those virtual addresses that are in system (S0) space.

Exit mechanism
The alternate start-I/O routine completes I/O requests by calling COM_STD$POST. This routine places each IRP in the I/O postprocessing queue and returns control to the driver. The driver can then fetch another IRP from an internal queue. If no IRPs remain, the driver returns control to EXE_STD$ALTQUEPKT, which relinquishes fork level synchronization and returns to the driver FDT routine that called it. The FDT routine performs any required postprocessing and returns the SS$_FDT_COMPL status to its caller.

Description

An alternate start-I/O routine initiates requests for activity on a device that can process two or more I/O requests simultaneously. Because the method by which the alternate start-I/O routine is invoked bypasses the unit's pending-I/O queue (UCB$L_IOQFL) and the device busy flag (UCB$V_BSY in UCB$L_STS), the routine is activated regardless of whether the device unit is busy with another request.

As a result, the driver that incorporates an alternate start-I/O routine must use its own internal I/O queues (in a UCB extension, for instance) and maintain synchronization with the unit's pending-I/O queue. In addition, if the routine processes more than one IRP at a time, it must use separate fork blocks for each request.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- You must indicate the entry point of the routine with a $DRIVER_ALTSTART_ENTRY macro, indicating which registers must be saved and restored across routine execution.
- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.
- You should examine the routine's use of suspension mechanisms (for instance, its forking, wait-for-interrupt, and resource-wait semantics) to determine whether it needs to be adapted to use the kernel process services. Typically a driver that makes subroutine calls before suspending itself (and relies on the previous context of these subroutines remaining intact on the stack), must be adapted to use the kernel process services.
- If the routine need not be converted to a kernel process, you should replace any calls to EXE$FORK, EXE$FORK_WAIT, EXE$IOFORK, IOC$WFIKPCH, IOC$WFIRLCH, IOC$REQPCHANH, and IOC$REQPCHANL with invocations of the appropriate suspension macro or with calls to EXE_STD$PRIMITIVE_FORK, IOC_STD$PRIMITIVE_WFIKPCH, IOC_STD$PRIMITIVE_WFIRLCH, IOC_STD$PRIMITIVE_REQCHANH, or IOC_STD$PRIMITIVE_REQCHANL.

Cancel-I/O Routine

Prevents further device-specific processing of the I/O request currently being processed on a device.

Format

CANCEL (chan, irp, pcb, ucb, reason)

Arguments

Argument	Type	Access	Mechanism
chan	integer	input	value
irp	IRP	input	reference
pcb	PCB	input	reference
ucb	UCB	input	reference
reason	integer	input	value

chan
Channel index number.

irp
I/O request packet, if any, for device (contents of UCB$L_IRP).

pcb
Process control block of process for which the I/O request is being canceled.

ucb
Unit control block.

reason
Reason for cancellation, one of the following:

CAN$C_CANCEL	Called by $CANCEL system service
CAN$C_DASSGN	Called by $DASSGN or $DALLOC system service

Essentials

Identifying the Routine

Supply the address of the cancel-I/O routine in the **cancel** argument of the DDTAB macro. The macro places the procedure value of this routine into DDT. Many drivers specify the system routine IOC_STD$CANCELIO as their cancel-I/O routine.

Declaring the Entry Point

Use:

```
$DRIVER_CANCEL_ENTRY [preserve=<R2,R3,R4>] [,fetch=YES]
```

where:

> **preserve** indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the cancel I/O routine
> **fetch**=**YES**, the default, loads the channel index number into R2, the cancellation reason into R8, and the addresses of the IRP, PCB, and UCB into R3, R4, and R5, respectively. **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver cancel-I/O routine that uses this macro can access any of its arguments by using a symbolic name of the form CANARG$_**argument-name**.

Called by

System routines call a driver's cancel-I/O routine under the following circumstances:

- When a process issues a Cancel-I/O-on-Channel system service ($CANCEL)
- When a process deallocates a device, causing the device's reference count (UCB$L_REFC) to become zero (that is, no process I/O channels are assigned to the device)
- When a process deassigns a channel from a device, using the $DASSGN system service
- When the command interpreter performs cleanup operations as part of image termination by canceling all pending I/O requests for the image and closing all image-related files open on process I/O channels

Context

A cancel-I/O routine begins execution at fork IPL, holding the corresponding fork lock. It must return control to its caller in this context.

A cancel-I/O routine executes in kernel mode in the context of the caller of the $CANCEL, $DALLOC, or $DASSGN system service.

Exit mechanism

The cancel-I/O routine returns to its caller.

Description

A driver's cancel-I/O routine must perform the following tasks:

1. Confirm that the device is busy by examining the device-busy bit in the UCB status longword (UCB$V_BSY in UCB$L_STS).
2. Confirm that the process ID (PID) of the request the device is servicing (IRP$L_PID) matches that of the process requesting the cancellation (PCB$L_PID).
3. Confirm that the channel-index number of the request the device is servicing (IRP$L_CHAN) matches that specified in the cancel-I/O request.
4. Cause to be completed (canceled) as quickly as possible all active I/O requests on the specified channel that were made by the process that has requested the cancellation. The cancel-I/O routine usually accomplishes this by setting UCB$V_CANCEL in the UCB$L_STS. When the next interrupt or timeout occurs for the device, the driver's start-I/O routine detects the presence of an active but canceled I/O request by testing this bit and takes appropriate action, such as completing the request without initiating any further device

activity. Other driver routines, such as the timeout handling routine, check the cancel-I/O bit to determine whether to retry the I/O operation or abort it.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- You must indicate the entry point of a cancel-I/O routine with a $DRIVER_CANCEL_ENTRY macro, indicating which registers must be saved and restored across routine execution.

Cancel Selective Routine

Performs additional processing on a list of I/O requests that have been canceled.

Format

status=CANCEL_SELECTIVE (pcb, ucb, chan, iosb_vector, iosb_count)

Arguments

Argument	Type	Access	Mechanism
pcb	PCB	input	reference
ucb	UCB	input	reference
chan	integer	input	value
iosb_vector	address	input	value
iosb_count	integer	input	value

pcb
Process control block of process for which the I/O request is being canceled.

ucb
Unit control block.

chan
Channel index number.

iosb_vector
Vector of address of I/O status blocks (IOSBs), or zero.

iosb_count
Number of addresses in the IOSB vector.

Essentials

Identifying the Routine

Supply the address of the cancel selective routine in the **cancel_selective** argument of the DDTAB macro. The macro places the procedure value of this routine into DDT.

Declaring the Entry Point

Use:

$DRIVER_CANCEL_SELECTIVE_ENTRY [preserve=<>] [,fetch=YES]

where:

> **preserve** indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the cancel selective routine.

fetch=**YES**, the default, loads SS$_UNSUPPORTED status into R0, the IOSB vector into R7, the IOSB count into R8, and the addresses of the PCB and UCB into R4 and R5, respectively, **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver cancel selective routine that uses this macro can access any of its arguments by using a symbolic name of the form CANSARG$_**argument-name**.

Called by

EXE$CANCEL_SELECTIVE calls a driver's cancel selective routine.

Context

A cancel selective routine is called at device IPL, holding the corresponding device lock and the appropriate fork lock. The channel control block (CCB) is locked in memory. It must return control to EXE$CANCEL_SELECTIVE in this context.

Exit mechanism

The cancel selective routine returns to its caller.

Description

Reserved to Digital.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note that you must indicate the entry point of a cancel-I/O routine with a $DRIVER_CANCEL_SELECTIVE_ENTRY macro, indicating which registers must be saved and restored across routine execution.

Channel Assign Routine

Performs specialized operations when a channel is assigned to a non-network device.

Format

CHANNEL_ASSIGN (ucb, ccb)

Arguments

Argument	Type	Access	Mechanism
ucb	UCB	input	reference
ccb	CCB	input	reference

ucb

Unit control block.

ccb

Channel control block.

Essentials

Identifying the Routine

Supply the address of the channel assign routine in the **channel_assign** argument of the DDTAB macro. The macro places the procedure value of this routine into DDT.

Declaring the Entry Point

Use:

```
$DRIVER_CHANNEL_ASSIGN_ENTRY [preserve=<>] [,fetch=YES]
```

where:

> **preserve** indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the cancel selective routine.
> **fetch**=**YES**, the default, loads the addresses of UCB and CCB into R5 and R8, respectively, **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver channel assign routine that uses this macro can access any of its arguments by using a symbolic name of the form CHANARG$_**argument-name**.

Called by

EXE$ASSIGN_LOCAL (in module SYSASSIGN) calls a driver's channel assign routine.

Context

A channel assign routine is called in kernel mode at IPL 0.

Exit mechanism
The channel assign routine returns to its caller.

Description

Reserved to Digital.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note that you must indicate the entry point of a channel assign routine with a $DRIVER_CHANNEL_ASSIGN_ENTRY macro, indicating which registers must be saved and restored across routine execution.

Cloned UCB Routine

Completes the initialization of the UCB cloned when a channel is requested for a template device.

Format

status = CLONEDUCB (cloned_ucb, ddt, pcb, template_ucb)

Arguments

Argument	Type	Access	Mechanism
cloned_ucb	UCB	input	reference
ddt	DDT	input	reference
pcb	PCB	input	reference
template_ucb	UCB	input	reference

cloned_ucb
Cloned unit control block. Fields of the cloned UCB have been initialized as follows:

Field	Value
UCB$L_FQFL	Address of UCB$L_FQFL
UCB$L_FQBL	Address of UCB$L_FQFL
UCB$L_FPC	0
UCB$Q_FR3	0
UCB$Q_FR4	0
UCB$W_BUFQUO	0
UCB$L_LINK	Address of next UCB in DDB chain
UCB$L_IOQFL	Address of UCB$L_IOQFL
UCB$L_IOQBL	Address of UCB$L_IOQFL
UCB$W_UNIT	Device unit number
UCB$W_CHARGE	Mailbox byte quota charge (UCB$W_SIZE)
UCB$L_REFC	0
UCB$L_STS	UCB$V_DELETEUCB set, UCB$V_ONLINE set
UCB$L_DEVSTS	UCB$V_DELMBX set if DEV$V_MBX is set in UCB$L_DEVCHAR

Field	Value
UCB$L_OPCNT	0
UCB$L_SVAPTE	0
UCB$L_BOFF	0
UCB$L_BCNT	0
UCB$L_ORB	Address of object rights block (ORB) for the cloned UCB

The cloned UCB ORB is initialized using the template UCB ORB. You can modify the ORB on the template UCB using the DCL SET SECURITY command.

ddt
Driver dispatch table.

pcb
Process control block of the current process.

template_ucb
Template unit control block.

Essentials

Identifying the Routine

Specify the address of a cloned UCB routine in the **cloneducb** argument of the DDTAB macro. The macro places the procedure value of the routine into the DDT. Only drivers for template devices, such as mailboxes, specify a cloned UCB routine.

Declaring the Entry Point

Use:

```
$DRIVER_CLONEDUCB_ENTRY [preserve=<R3>] [,fetch=YES]
```

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the cloned UCB routine
fetch=**YES**, the default, loads SS$_NORMAL status into R0, and the addresses of the cloned UCB, DDT, PCB, and template UCB into R2, R3, R4, and R5, respectively; **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver cloned UCB routine that uses this macro can access any of its arguments by using a symbolic name of the form CLONEARG$_**argument-name**.

Called by

EXE$ASSIGN calls the driver's cloned UCB routine when an Assign I/O Channel system service request ($ASSIGN) specifies a template device (that is, bit UCB$V_TEMPLATE in UCB$L_STS is set).

Context

A cloned UCB routine executes at IPL$_ASTDEL, holding the I/O database mutex (IOC$GL_MUTEX).

A cloned UCB routine executes in kernel mode in the context of the process that called the $ASSIGN system service.

Exit mechanism

A cloned UCB routine must return control and status to EXE$ASSIGN. If the routine returns error status in R0, EXE$ASSIGN undoes the process of UCB cloning and completes with failure status in R0.

Description

When a process requests that a channel be assigned to a template device, EXE$ASSIGN does not assign the channel to the template device itself. Rather, it creates a copy of the template device's UCB and ORB, initializing and clearing certain fields as appropriate.

The driver's cloned UCB routine verifies the contents of these fields and completes their initialization.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note that you must indicate the entry point of a cloned UCB with a $DRIVER_CLONEDUCB_ENTRY macro, indicating which registers must be saved and restored across routine execution.

Controller Initialization Routine

Prepares a controller for operation.

Format

status = CTRLINIT (idb, ddb, crb)

Arguments

Argument	Type	Access	Mechanism
idb	IDB	input	reference
ddb	DDB	input	reference
crb	CRB	input	reference

idb
Interrupt dispatch block associated with the controller.

ddb
Device data block associated with the controller.

crb
Controller request block.

Essentials

Identifying the Routine

Specify the address of a controller initialization routine in the **ctrlinit** argument of the DDTAB macro. The macro places the procedure value of this routine into the DDT.

Declaring the Entry Point

Use:

$DRIVER_CTRLINIT_ENTRY [preserve=<R2>] [,fetch=YES]

where:

> **preserve** indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the controller initialization routine.
> **fetch**=**YES**, the default, loads SS$_NORMAL status into R0, the address of the IDB into R4 and R5, and the addresses of the DDB and CRB into R6 and R8, respectively; **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver controller initialization routine that uses this macro can access any of its arguments by using a symbolic name of the form CTRLARG$_**argument-name**.

Called by

The driver-loading procedure calls a driver's controller initialization routine when processing a CONNECT command. Also, the system calls this routine if the device, controller, processor, or adapter to which the device is connected experiences a power failure.

Context

OpenVMS calls a controller initialization routine at IPL$_POWER. If it must lower IPL, the controller initialization routine cannot explicitly do so. Rather, it must fork. Because the driver-loading procedure calls the unit initialization routine immediately after the controller initialization returns control to it, the driver's initialization routines must synchronize their activities. If the controller initialization routine forks, the unit initialization routine must be prepared to execute before the controller initialization routine completes.

The portion of the controller initialization that services power failure cannot acquire any spin locks. As a result, the routine cannot fork to perform power failure servicing.

Because a controller initialization routine executes within system context, it can refer only to those virtual addresses that reside in system (S0) space.

Exit mechanism

The controller initialization routine returns success or failure status to its caller.

Description

Some controllers require initialization when the system's driver-loading routine loads the driver and when the system is recovering from a power failure. Depending on the device, a controller initialization routine performs any and all of the following actions:

- Determines whether it is being called as a result of a power failure by examining the power bit (UCB$V_POWER in UCB$L_STS) in the UCB. A controller initialization routine may want to perform or avoid specific tasks when servicing a power failure.
- Clears error-status bits in device registers.
- Enables controller interrupts.
- Allocates resources that must be permanently allocated to the controller.
- If the controller is dedicated to a single-unit device, such as a printer, fills in IDB$PS_OWNER and set the online bit (UCB$V_ONLINE in UCB$L_STS).
- Initializes the interrupt vectors of devices with programmable interrupt vectors.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- You must indicate the entry point of the routine with a $DRIVER_CTRLINIT_ENTRY macro to indicate which registers must be saved and restored across routine execution.

- An OpenVMS VAX device driver specifies a controller initialization routine by invoking the DPT_STORE macro to place its procedure value into the interrupt transfer vector block (CRB$L_INTD+VEC$L_INITIAL). An OpenVMS Alpha device driver specifies the routine in the **ctrlinit** argument of the DDTAB macro.
- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.
- The controller initialization routine of an OpenVMS VAX device driver receives the addresses of the device CSR in R4 and the IDB in R5. An OpenVMS Alpha device driver's controller initialization routine is not passed the address of the CSR. It may access the controller's register by means of the controller register access mailbox (CRAM), the address of which is provided in IDB$PS_CRAM.
- A controller initialization routine that must initialize the programmable interrupt vectors of a device does so by referring to the vector offset placed in IDB$L_VECTOR by the driver-loading procedure. For a device with multiple interrupt vectors, IDB$L_VECTOR contains the address of a vector list extension (VLE) which contains a list of vector offsets.
- An OpenVMS Alpha controller initialization routine must return success or failure status to its caller.

Driver Channel Grant Fork Routine Entry

Enabled via the IOC_STD$REQCHANx or IOC$REQCHANx routines if the CRB is not immediately available. The procedure value of the grant routine is contained in ucb->ucb$l_fpc. The grant routine is invoked by IOC_STD$RELCHAN which has been enhanced to support both the JSB interface and the new standard call interface. The above also applies to IOC$RELCHAN which is now simply a JSB-to-CALL interface jacket routine around IOC_STD$RELCHAN.

Description

The JSB interface for the channel grant routine is:

```
JSB driver_channel_grant_routine

       Inputs:

         R3            contains a pointer to the IRP,

         R4            contains a pointer to the IDB,

         R5            contains a pointer to the UCB.

       Outputs:

         R0-R5         may be scratched by the routine.
```

The standard call interface for the channel grant routine is:

```
       void driver_channel_grant_routine (IRP *irp, IDB *idb, UCB *ucb);

       Inputs:

         irp           is a pointer to the IRP,

         idb           is a pointer to the IDB,

         ucb           is a pointer to the UCB.
```

Driver Device Timeout Routine Entry

Enabled by the WFIKPCH or WFIRLCH macros and invoked by the EXE$TIMEOUT routine. The EXE$TIMEOUT routine supports both timeout routines using the JSB interface and the standard call interface.

Description

The JSB interface for the interrupt timeout routine is:

```
JSB driver_timeout_routine
Inputs:
  R3           contains a pointer to the IRP from UCB$Q_FR3(R5),
  R4           contains the 64-bit value from UCB$Q_FR4(R5),
  R5           contains a pointer to the UCB.
Outputs:
  R0-R4        may be scratched by the routine.
```

The standard call interface for the interrupt timeout routine is:

```
void driver_timeout_routine (IRP *irp, int64 fr4, UCB *ucb);
Inputs:
  irp          is a pointer to the IRP from ucb->ucb$q_fr3,
  fr4          is the 64-bit value from ucb->fkb$q_fr4,
  ucb          is a pointer to the UCB,
```

The procedure value of the driver interrupt timeout routine is found in ucb->ucb$ps_toutrout.

Note

By default the WFIKPCH macro and the IOC$PRIMITIVE_WFIKPCH JSB interface routine set the ucb$ps_toutrout cell to contain the same value as ucb$l_fpc.

Driver Resume from Interrupt Routine Entry

The driver resume from interrupt routine is setup by the WFIKPCH macro and is invoked by the driver's interrupt service routine.

Description

The JSB interface for the driver interrupt resume routine is:

```
JSB driver_resume_routine
Inputs:
  R3             contains a pointer to the IRP from UCB$Q_FR3(R5),
  R4             contains the 64-bit value from UCB$Q_FR4(R5),
  R5             contains a pointer to the UCB.
Outputs:
  R0-R4          may be scratched by the routine.
```

The recommended standard call interface for the driver resume from interrupt routine is:

```
void driver_resume_routine (IRP *irp, int64 fr4, UCB *ucb);
Inputs:
  irp            is a pointer to the IRP from ucb->ucb$q_fr3,
  fr4            is the 64-bit value from ucb->fkb$q_fr4,
  ucb            is a pointer to the UCB,
```

Note

The resume from interrupt routine interface must conform exactly to the calling convention used in the interrupt service routine in that driver. This differs from other routines, for example the interrupt timeout routine, which could be written to use either the traditional or the new interface.

It may be possible to eliminate the driver resume from interrupt routine by moving some processing directly into the interrupt service routine and by resuming the driver in a fork routine.

Start I/O Routine (Simple Fork, JSB Environment)

Activates a device to process a requested I/O function.

Driver Unloading Routine

******Not supported in OpenVMS Alpha drivers********

FDT Upper-Level Action Routine

Performs any device-dependent activities needed to prepare the I/O database to process an I/O request.

Format

status = driver_FDT_routine (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism
irp	IRP	input	reference
pcb	PCB	input	reference
ucb	UCB	input	reference
ccb	CCB	input	reference

irp
I/O request packet for the current I/O request. An FDT routine may read the following IRP fields:

Field	Contents
IRP$L_FUNC	I/O function code supplied in the $QIO request
IRP$L_QIO_P*n*	Function-specific $QIO system service arguments (**p1** through **p6**); *n* corresponds to an integer from 1 to 6.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Essentials

Identifying the Routine
Use the FDT_ACT macro to insert the procedure value of an upper-level FDT action routine into the FDT action routine vector slot that corresponds to a specified I/O function code.

Declaring the Entry Point
Use:

```
$DRIVER_FDT_ENTRY
[preserve=<R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15>] [,fetch=YES]
```

where:

> **preserve** indicates the registers to be preserved (in addition to those automatically preserved by the MACRO compiler) across the call to the upper-level FDT action routine.
> **fetch**=**YES**, the default, loads the addresses of the IRP, PCB, UCB, and CCB into R3, R4, R5, and R6, respectively; **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver upper-level FDT action routine that uses this macro can access any of its arguments by using a symbolic name of the form FDTARG$_**argument-name**.

Called by

The $QIO system service calls a driver's upper-level FDT action routine from the module SYSQIOREQ. An upper-level FDT action routine can call any number of FDT support routines, as long as each routine returns control and status to the upper-level routine.

Context

An FDT routine is called at IPL$_ASTDEL and must exit at IPL$_ASTDEL. An FDT routine must not lower IPL below IPL$_ASTDEL. If it raises IPL, it must lower it to IPL$_ASTDEL before passing control to any other code. Similarly, before exiting, it must release any spin locks it may have acquired in an OpenVMS multiprocessing environment.

FDT routines execute in the context of the process that requested the I/O activity. If an FDT routine alters the stack, it must restore the stack before returning control to the caller of the routine.

Exit mechanism

An FDT routine must return control and status to its caller. An upper-level FDT action routine returns SS$_FDT_COMPL status to the $QIO system service and passes the return status to be delivered to the caller of $QIO in the FDT_CONTEXT structure.

Description

An upper-level FDT routine (and any FDT support routine it may call) validates the function-dependent arguments to a $QIO system service request and prepares the I/O database to service the request. For each function that a device supports, an upper-level FDT action routine must provide preprocessing of requests for that function. FDT processing may complete a function that does not involve an I/O transfer. Otherwise FDT processing can abort the request or deliver it to the driver.

An OpenVMS Alpha upper-level FDT action routine can invoke the $FDTARGDEF macro, defined in SYS$LIBRARY:LIB.MLB, to provide symbolic names for the standard AP offsets of the four parameters provided as input (IRP, PCB, UCB, and CCB) to all upper-level FDT action routines. A routine that does so can use names of the form FDTARG$_*xxx*, where *xxx* is the 3-letter structure acronym, to access the input parameters.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- You must indicate the entry point of each upper-level FDT action routine with a $DRIVER_FDT_ENTRY macro, indicating which registers must be saved and restored across routine execution.
- You should examine an FDT routine's use of R0, R7, and R8.

  The FDT routine of an OpenVMS VAX device driver may obtain the address of FDT routine being called from R0, the number of the bit that specifies the code for the requested I/O function from R7, and the address of the entry in the function decision table that dispatched control to this FDT routine.

  An OpenVMS Alpha driver can obtain the user-supplied function code from IRP$L_FUNC. It can obtain the address of the start of the FDT from DDT$PS_FDT2. The DDT address is available from UCB$L_DDT.
- An FDT routine of an OpenVMS VAX device driver accesses values of the function-dependent arguments specified in the $QIO request as offsets from the value of the AP; an OpenVMS Alpha device driver obtains them from the IRP (at symbolic offsets IRP$L_QIO_P1 through IRP$L_QIO_P6).

FDT Error-Handling Callback Routine

Processes error conditions that occur during EXE_STD$READLOCK, EXE_STD$WRITELOCK, and EXE_STD$MODIFYLOCK processing.

Format

status = error_callback (irp, pcb, ucb, ccb, status)

Arguments

Argument	Type	Access	Mechanism
irp	IRP	input	reference
pcb	PCB	input	reference
ucb	UCB	input	reference
ccb	CCB	input	reference
status	integer	input	value

irp
I/O request packet for the current I/O request.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

status
Error status returned by buffer accessibility check (SS$_ACCVIO or SS$_BADPARAM) or buffer locking operation (SS$_ACCVIO, SS$_INSFWSL, or page fault status).

Essentials

Identifying the Routine
Use the **errtn** argument in a call to EXE_STD$MODIFYLOCK, EXE_STD$READLOCK, or EXE_STD$WRITELOCK.

Declaring the Entry Point
Use:

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

where:

preserve indicates the registers to be preserved (in addition to those automatically preserved by the MACRO compiler) across the call to the upper-level FDT action routine.
fetch=**YES**, the default, loads the addresses of the IRP, PCB, UCB, CCB, and status into R3, R4, R5, R6, and R0, respectively; **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver error-handling callback routine that uses this macro can access any of its arguments by using a symbolic name of the form ERRARG$_**argument-name**.

Called by

EXE_STD$MODIFYLOCK, EXE_STD$READLOCK, and EXE_STD$WRITELOCK call the driver's error-handling callback routine to process errors incurred by a buffer accessibility check or buffer locking operation.

Context

An error-handling callback routine is called at IPL$_ASTDEL and must exit at IPL$_ASTDEL. An error-handling callback routine must not lower IPL below IPL$_ASTDEL. If it raises IPL, it must lower it to IPL$_ASTDEL before passing control to any other code. Similarly, before exiting, it must release any spin locks it may have acquired in an OpenVMS multiprocessing environment.

Error-handling callback routines execute in the context of the process that requested the I/O activity. If a routine alters the stack, it must restore the stack before returning control to the caller of the routine.

Exit mechanism

An error-handling callback routine must return control to its caller and preserve the contents of R0 and R1.

Description

An error-handling callback routine processes any errors incurred by a call to EXE_STD$MODIFYLOCK, EXE_STD$READLOCK, or EXE_STD$WRITELOCK.

A driver typically requires an error-handling callback routine if it must lock multiple areas into memory for a single I/O request and must regain control, if the request is to be aborted, to unlock these areas. The routine performs such operations as locating the addresses of the previously allocated buffers (typically stored in the IRP) and calling MMG_STD$UNLOCK to release them.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- You must indicate the entry point of each FDT error-handling callback routine with a $DRIVER_ERRRTN_ENTRY macro, indicating which registers must be saved and restored across routine execution.
- You should examine an FDT routine's use of R0, R7, and R8.

  The FDT routine of an OpenVMS VAX device driver may obtain the address of FDT routine being called from R0, the number of the bit that specifies the code for the requested I/O function from R7, and the address of the entry in the function decision table that dispatched control to this FDT routine.

An OpenVMS Alpha driver can obtain the user-supplied function code from IRP$L_FUNC. It can obtain the address of the start of the FDT from DDT$PS_FDT2. The DDT address is available from UCB$L_DDT.

- An FDT routine of an OpenVMS VAX device driver accesses values of the function-dependent arguments specified in the $QIO request as offsets from the value of the AP; an OpenVMS Alpha device driver obtains them from the IRP (at symbolic offsets IRP$L_QIO_P1 through IRP$L_QIO_P6).

Interrupt Service Routine

Processes interrupts generated by a device. The Interrupt Service routine is called by the system interrupt dispatcher.

Format

DRIVER_INTERRUPT (idb, scb_offset)

Arguments

Argument	Type	Access	Mechanism
idb	IDB	input	reference
scb_offset	integer	input	value

idb

Interrupt dispatch block.

Essentials

Identifying the Routine

Devices require an interrupt service routine for each interrupt vector. Use the DPT_STORE_ISR macro to store the ISR procedure descriptor and entry point address in the interrupt transfer vector block (VEC) at CRB$L_INTD. You can find the second and third VECs at CRB$L_INTD2 and CRB$L_INTD+2*VEC$K_LENGTH, respectively.

Declaring the Entry Point

Indicate the entry point of an OpenVMS Alpha interrupt service routine with a .CALL_ENTRY MACRO-32 compiler directive to indicate which registers are provided as input or used as output and which must be saved and restored. If the interrupt service routine forks, transferring control to a fork routine, it must declare, at its .CALL_ENTRY point, R3, R4, and R5 as **input** registers.

Called by

The interrupt service routine is called either by the OpenVMS interrupt dispatcher (for direct-vectored adapters) or by an adapter interrupt service routine (for non-direct-vector adapters).

Context

An OpenVMS Alpha driver's interrupt service routine conforms to the OpenVMS calling standard.

An interrupt service routine is called, executes, and returns at device IPL. It must obtain the device lock associated with its device IPL. It performs this acquisition as soon as it obtains the address of the UCB of the interrupting device. It must release this device lock before dismissing the interrupt.

At the execution of a driver's interrupt service routine, the processor is running in interrupt mode on the kernel stack. As a result, an interrupt service routine can reference only those virtual addresses that reside in system (S0) space.

Resuming the Suspended Driver Thread

The method that an interrupt service routine should use to invoke the driver's resume from interrupt routine depends on how the driver suspended its execution.

If the driver is using the simple fork mechanism with a JSB-based environment then the driver resume from interrupt routine is invoked by the following:

```
MOVX    UCB$Q_FR3(R5),R3  ;R3 = FR3 (64-bits)
MOVX    UCB$Q_FR4(R5),R4  ;R4 = FR4 (64-bits)
JSB     @UCB$L_FPC(R5)
```

If the driver is using the simple fork mechanism with a CALL-based environment then the driver resume from interrupt routine is invoked in C by the following:

```
(ucb->ucb$l_fpc)( ucb->ucb$q_fr3, ucb->ucb$q_fr4, ucb);
```

or in MACRO-32 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)
```

If the driver is using the kernel process mechanism then the suspended kernel process can be resumed in C by the following:

```
exe$kp_restart( kpb );
```

or:

```
(ucb->ucb$l_fpc)( ucb->ucb$q_fr3, ucb->ucb$q_fr4, ucb);
```

or in MACRO-32 by the following:

```
PUSHL   UCB$Q_FR4(R5)     ;Param1 = KPB address
CALLS   #1,EXE$KP_RESTART
```

or:

```
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)
```

Exit mechanism

The interrupt service routine returns to the interrupt dispatcher with a RET instruction.

Description

An interrupt service routine performs the following functions:

1. Determines whether the interrupt is expected.
2. Processes or dismisses unexpected interrupts.
3. Activates the suspended driver so it can process expected interrupts.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- An OpenVMS VAX device driver declares an interrupt service routine by issuing the DPT_STORE macro to store its address in an interrupt transfer vector block. Because the OpenVMS Alpha interrupt dispatcher requires the addresses of both the code entry point and the procedure descriptor of an interrupt service routine, you must use the new DPT_STORE_ISR macro (which generates both) to declare the routine.
- The OpenVMS VAX interrupt dispatcher issues a JSB instruction to pass control to an OpenVMS VAX driver's interrupt service routine; the OpenVMS Alpha interrupt dispatcher issues a standard call to a driver's interrupt service routine. This results in some substantial differences:
    - You must indicate the entry point of an OpenVMS Alpha interrupt service routine with a .CALL_ENTRY MACRO-32 compiler directive to indicate which registers are provided as input or used as output and which must be saved and restored.
    - An OpenVMS VAX driver's interrupt service routine must preserve any of the non-scratch registers R2 through R15 if it uses them.
    - An OpenVMS VAX driver's interrupt service routine is passed various information on the stack, including the address of the IDB, the contents of R0 through R5, the PC, and PSL at the time of the interrupt.

      The only parameter passed to an OpenVMS Alpha driver's interrupt service routine is the address of the IDB (that is, the contents of VEC$L_IDB). The routine cannot reference data on the stack.
    - Before exiting, an OpenVMS VAX driver's interrupt service routine removes the address of the pointer to the IDB from the top of the stack and restores the registers OpenVMS saved when dispatching the interrupt.

      An OpenVMS Alpha driver's interrupt service routine does not perform these actions.
    - An OpenVMS VAX driver's interrupt service routine exits with an REI instruction.

      An OpenVMS Alpha driver's interrupt service routine exits by returning control with a RET instruction.
- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.
- If you alter the driver's suspension mechanism such that it uses the OpenVMS kernel process services, you must change the mechanism by which the interrupt service routine reactivates lower IPL execution threads by replacing the IOFORK macro with the KP_STALL_IOFORK macro.

Mount Verification Routine

Performs device-specific mount verification.

Format

MNTVER (irp, ucb)

Arguments

Argument	Type	Access	Mechanism
irp	IRP	input	reference
ucb	UCB	input	reference

irp
I/O request packet, or zero to complete mount verification.

ucb
Unit control block.

Essentials

Identifying the Routine
Supply the address of the mount verification routine in the **mntver** argument of the DDTAB macro. The macro places the procedure value of this routine into DDT. The default value of this argument, IOC_STD$MNTVER, is the only value allowed for device drivers not supplied by Digital.

Declaring the Entry Point
Use:

$DRIVER_MNTVER_ENTRY [preserve=<>] [,fetch=YES]

where:

> **preserve** indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the cancel selective routine.
> **fetch**=**YES**, the default, loads the addresses of IRP and UCB into R3 and R5, respectively, **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver mount verification routine that uses this macro can access any of its arguments by using a symbolic name of the form MNTARG$_**argument-name**.

Called by
Routine DRIVER_CODE in module MOUNTVER calls a driver’s mount verification routine.

Context
A mount verification routine is called at fork IPL with the corresponding fork lock held in a multiprocessing system.

Exit mechanism
The mount verification routine returns to its caller.

Description

Reserved to Digital.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note that you must indicate the entry point of a mount verification routine with a $DRIVER_MNTVER_ENTRY indicating which registers must be saved and restored across routine execution.

Register Dumping Routine

Copies the contents of a device's registers to an error message buffer or a diagnostic buffer.

Format

status = REGDMP (buffer, arg_2, ucb)

Arguments

Argument	Type	Access	Mechanism
buffer	address	input	reference
arg_2	unspecified	input	reference
ucb	UCB	input	reference

buffer
Address of buffer into which a register dumping routine copies the contents of device registers.

arg_2
Device-specific argument, usually a controller register access mailbox (CRAM).

ucb
Unit control block.

Essentials

Identifying the Routine
Specify the name of the register dumping routine in the **regdmp** argument of the DDTAB macro. This macro places the procedure value of the routine into the DDT.

Declaring the Entry Point
Use:

$DRIVER_REGDUMP_ENTRY [preserve=<R2>] [,fetch=YES]

where:

> **preserve** indicates the registers to be preserved (in addition to those automatically preserved by the MACRO compiler) across the call to the register dumping routine.
> **fetch**=**YES**, the default, loads the addresses of the buffer, the driver-specific argument. and the UCB into R0, R4, and R5, respectively; **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver register dumping routine that uses this macro can access any of its arguments by using a symbolic name of the form REGARG$_**argument-name**.

Called by

The system error-logging routines (ERL_STD$DEVICERR, ERL_STD$DEVICTMO, and ERL_STD$DEVICEATTN) and diagnostic buffer filling routine (IOC_STD$DIAGBUFILL) call the register dumping routine.

Context

OpenVMS calls a register dumping routine at the same interrupt service routine (IPL) at which the driver called the OpenVMS Alpha system routine ERL_STD$DEVICERR, ERL_STD$DEVICTMO, ERL_STD$DEVICEATTN, or IOC_STD$DIAGBUFILL. A register dumping routine must not change IPL.

A register dumping routine executes within the context of an IPL routine or a driver fork process, using the kernel-mode stack. As a result, it can only refer to those virtual addresses that reside in system (S0) space. If it uses the stack, the register dumping routine must restore the stack before passing control to another routine, waiting for an interrupt, or returning control to its caller.

Exit mechanism

The register dumping routine returns to its caller.

Description

A register dumping routine fills the indicated buffer as follows:

1. Writes a longword value representing the number of device registers to be written into the buffer
2. Moves device register longword values into the buffer following the register count longword

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- You must indicate the entry point of the routine with a $DRIVER_REGDUMP_ENTRY macro, indicating which registers must be saved and restored across routine execution.
- An OpenVMS VAX device driver's register dumping routine is passed the address of the device's CSR in R4 (if the driver invoked the WFIKPCH macro to wait for an interrupt or timeout).

  An OpenVMS Alpha device driver's register dumping routine is not passed the address of the CSR. It may access the controller's register by means of the controller register access mailbox (CRAM), the address of which is usually passed in **arg_2**.
- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.

Start-I/O Routine (Simple Fork, Call Environment)

Activates a device to process a requested I/O function.

Format

START (irp, ucb)

Arguments

Argument	Type	Access	Mechanism
irp	IRP	input	reference
ucb	UCB	input	reference

irp

I/O request packet.

ucb

Unit control block. The start-I/O routine uses information from the following UCB fields to calculate the size and location of a transfer:

Field	Description
UCB$L_BCNT	Number of bytes to be transferred, copied from the low-order word of IRP$L_BCNT
UCB$L_BOFF	Byte offset into first page of direct-I/O transfer; for buffered-I/O transfers, number of bytes to be charged to the process allocating the buffer
UCB$L_SVAPTE	For a *direct-I/O* transfer, virtual address of first page-table entry (PTE) of I/O-transfer buffer; for *buffered-I/O* transfer, address of buffer in system address space

Essentials

Identifying the Routine

Specify the name of the start-I/O routine in the **start** argument of the DDTAB macro. This macro places the address of the routine into the DDT.

Declaring the Entry Point

Use:

```
$DRIVER_START_ENTRY [preserve=<R2,R4>] [,fetch=YES]
```

where:

> **preserve** indicates the registers to be preserved (in addition to those automatically preserved by the MACRO compiler) across the call to the start-I/O routine

fetch=**YES**, the default, loads the addresses of the IRP and UCB into R3 and R5, respectively; **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver start-I/O routine that uses this macro can access any of its arguments by using a symbolic name of the form STARTARG$_**argument-name**.

Called by

A traditional start-I/O routine is called as the result of a standard call issued by IOC_STD$INITIATE and IOC_STD$REQCOM in module IOSUBNPAG.

Context

A start-I/O routine is placed into execution at fork IPL, holding the associated fork lock. It must relinquish control of the processor in the same context.

For many devices, the start-I/O routine raises IPL to IPL$_POWER to check that a power failure has not occurred on the device prior to loading the device's registers. The start-I/O routine initiates device activity at device IPL, after acquiring the corresponding device lock. An invocation of the WFIKPCH or WFIRLCH macro (or KP_STALL_WFIKPCH or KP_STALL_WFIRLCH) to wait for a device interrupt releases this device lock.

Because a start-I/O routine gains control of the processor in the context of a fork process, it can refer only to those addresses that reside in system (S0) space. If the start-I/O routine uses the stack, it must restore the stack before completing the request, waiting for an interrupt, or requesting system resources.

Exit mechanism

A traditional start-I/O routine suspends itself whenever it must wait for a required resource, such as a controller data channel. To do so, it invokes an OpenVMS macro (such as REQPCHAN) that saves its context in the UCB fork block, places the UCB in a resource wait queue, and returns control to the caller of the start-I/O routine.

The start-I/O routine also suspends itself when it issues a WFIKPCH or WFIRLCH macro to initiate device activity. These macros also store the driver's context in the UCB fork block to be restored when the device interrupts or times out.

The start-I/O routine is again suspended if it forks to complete servicing of a device interrupt. The IOFORK macro places driver context in the UCB fork block, inserts the fork block into a processor-specific fork queue, and requests a software interrupt from the processor at the corresponding fork IPL. After issuing an IOFORK macro, the routine returns control to the driver's interrupt service routine.

The routine completes the processing of an I/O request by invoking the REQCOM macro. In addition to initiating device-independent postprocessing of the current request, the REQCOM macro attempts to start the next request waiting for a device unit. If there are no waiting requests, the macro returns control to the caller of the start-I/O routine, which is the OpenVMS fork dispatcher.

Description

A driver's start-I/O routine activates a device and waits for a device interrupt or timeout. After a device interrupt, the driver's interrupt service routine returns control to the start-I/O routine at device IPL, holding the associated device lock.

The start-I/O routine usually forks at this time to perform various device-dependent postprocessing tasks, and returns control to the interrupt service routine.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- You must indicate the entry point of the start-I/O routine with a $DRIVER_START_ENTRY macro, indicating which registers must be saved and restored across routine execution.
- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access by means of a CRAM.
- You should examine the routine's use of suspension mechanisms (for instance, its forking, wait-for-interrupt, and resource-wait semantics) to determine whether it needs to be adapted to use the kernel process services. Typically a driver that makes subroutine calls before suspending itself (and relies on the previous context of these subroutines remaining intact on the stack), must be adapted to use the kernel process services.

Start-I/O Routine (Kernel Process)

Activates a device to process a requested I/O function.

Format

START (kpb)

Arguments

Argument	Type	Access	Mechanism
kpb	KPB	input	reference

kpb
Kernel process block.

Essentials

Identifying the Routine

Specify the name of the kernel process start-I/O routine (EXE_STD$KP_STARTIO) in the **start** argument of the DDTAB macro, and the name of the driver's start-I/O routine in the **kp_startio** argument.

Declaring the Entry Point

Indicate the entry point of a kernel process start-I/O routine with a .CALL_ENTRY MACRO-32 compiler directive to indicate which registers are provided as input or used as output and which registers must be saved and restored.

Called by

A kernel-process start-I/O routine is called by EXE_STD$KP_STARTIO in module KERNEL_PROCESS.

Context

A kernel process start-I/O routine is placed into execution at fork IPL, holding the associated fork lock. The kernel process start-I/O routine must relinquish control of the processor in the same context.

For many devices, the start-I/O routine raises IPL to IPL$_POWER to check that a power failure has not occurred on the device prior to loading the device's registers. The start-I/O routine initiates device activity at device IPL, after acquiring the corresponding device lock. An invocation of the KP_STALL_WFIKPCH or KP_STALL_WFIRLCH macro to wait for a device interrupt releases this device lock.

Because a start-I/O routine gains control of the processor in the context of a fork process, it can refer only to those addresses that reside in system (S0) space.

Neither the start-I/O routine that initiates a kernel process nor the kernel process thread can depend on inheriting the synchronization capabilities (such as spin locks and IPL) of the other when control is exchanged betwen them. If they must share data or perform other operations that require synchonization, they must explicitly establish a synchronization mechanism.

The kernel process cannot assume that its initiator is not running in parallel, nor can the initiator of the kernel process assume that the kernel process has already executed when EXE$KP_START returns control.

Exit mechanism
A kernel process start-I/O routine suspends itself whenever it must wait for a required resource, such as a controller data channel. To do so, the kernel process start-I/O routine invokes an OpenVMS macro (such as KP_STALL_REQCHAN) that saves its context in the UCB fork block, places the UCB in a resource wait queue, and returns control to the caller of the start-I/O routine.

The start-I/O routine also suspends itself when it issues a KP_STALL_WFIKPCH or KP_STALL_WFIRLCH macro to initiate device activity. These macros also store the driver's context in the UCB fork block to be restored when the device interrupts or times out.

The start-I/O routine is again suspended if it forks to complete servicing of a device interrupt. The KP_STALL_IOFORK macro places driver context in the UCB fork block, inserts the fork block into a processor-specific fork queue, and requests a software interrupt from the processor at the corresponding fork IPL. After issuing a KP_STALL_IOFORK macro, the routine issues an RSB instruction, returning control to the driver's interrupt service routine.

The routine completes the processing of an I/O request by invoking the KP_REQCOM macro. In addition to initiating device-independent postprocessing of the current request, the KP_REQCOM macro also attempts to start the next request waiting for a device unit. If there are no waiting requests, the macro returns control to the caller of the kernel process start-I/O routine, EXE$KP_STARTIO.

Description

A driver's start-I/O routine activates a device and waits for a device interrupt or timeout. After a device interrupt, the driver's interrupt service routine returns control to the start-I/O routine at device IPL, holding the associated device lock.

The start-I/O routine usually forks at this time to perform various device-dependent postprocessing tasks, and returns control to the interrupt service routine.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- If the routine need not be converted to a kernel process, you must replace any calls to EXE$FORK, EXE$FORK_WAIT, EXE$IOFORK, IOC$WFIKPCH, IOC$WFIRLCH, IOC$REQPCHANH, and IOC$REQPCHANL with invocations of the appropriate suspension macro or with calls to EXE_STD$PRIMITIVE_FORK, IOC_STD$PRIMITIVE_WFIKPCH, IOC_STD$PRIMITIVE_WFIRLCH, IOC_STD$PRIMITIVE_REQCHANH, or IOC_STD$PRIMITIVE_REQCHANL.
- You must indicate the entry point of a kernel process start-I/O routine with a .CALL_ENTRY MACRO-32 compiler directive to indicate which registers are provided as input or used as output and which registers must be saved and restored. A kernel process start-I/O routine invokes the KP_REQCOM macro (in place of the REQCOM macro) to return control properly to its caller.

Timeout Handling Code (Traditional)

Takes whatever action is necessary when a device has not yet responded to a request for device activity, and the time allowed for a response has expired.

Format

BNEQ timeout-code-address

Input

Location	**Contents**
R3	Contents of R3 when the last invocation of WFIKPCH or WFIRLCH occurred (usually the address of the IRP)
R4	Contents of R4 when the last invocation of WFIKPCH or WFIRLCH occurred (usually the address of the IDB)
R5	Address of UCB of the device
UCB$L_STS	UCB$V_INT and UCB$V_TIM clear; UCB$V_TIMOUT set

Essentials

Identifying the Timeout Handler

Specify the address of timeout code in the **excpt** argument to the WFIKPCH or WFIRLCH macro.

Branched to

The WFIKPCH and WFIRLCH macros use this entry point, but only when the label of timeout code is provided in their **excpt** argument. These macros are used in the driver's start-I/O routine; thus, strictly speaking, the driver itself is the only entity that uses this entry point.

The OpenVMS Alpha software timer interrupt service routine restarts a stalled driver fork procedure, passing to it a status (UCB$V_TIMOUT in UCB$L_STS) which is inspected by one of two instructions left at the top of the fork procedure by the WFIKPCH or WFIRLCH macro. If UCB$V_TIMOUT is set, the second instruction branches to the timeout code.

Context

Timeout code receives control at device IPL and must exit at device IPL. At the time the timeout code executes, the processor holds both the fork lock and the device lock associated with the device.

After taking whatever device-specific action is necessary at device IPL, timeout code can lower IPL to fork IPL to perform less critical activities. Because its caller restores IPL to fork IPL (and releases the device lock), if a timeout handler lowers IPL, it can do so only by forking or by performing the following steps:

1. Issue a DEVICEUNLOCK macro to lower to fork level
2. Perform timeout handling activities possible at the lower IPL

3. Issue a DEVICELOCK macro to again obtain the device lock and raise to device IPL

Timeout code can access only those virtual addresses that refer to system (S0) space.

Traditional timeout code can use R0, R1, and R2 freely, but must preserve the contents of all other registers. If it uses the stack, it must restore the stack before completing or canceling the current I/O request, waiting for an interrupt, or returning control to its caller.

Exit mechanism

Traditional timeout code issues an RSB instruction to return to the software timer interrupt service routine, restarts the I/O request, or invokes the REQCOM macro to complete the I/O request that encountered the timeout.

Description

There are no outputs required from timeout code but, depending on the characteristics of the device, timeout code might cancel or retry the current I/O request, send a message to the operator, or take some other action.

Before timeout code executes, the system has placed the device in a state in which no interrupt is expected (by clearing the bit UCB$V_INT in field UCB$L_STS). If the requested interrupt occurs while this routine executes, it will appear to be an unsolicited interrupt. Many drivers handle this situation by disabling interrupts while timeout code executes.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- On OpenVMS VAX systems, the software timer interrupt service routine issues a JSB instruction to a timeout handling routine within a driver when it detects that a device has timed out. On OpenVMS Alpha systems, the OpenVMS Alpha suspension macros provide a mechanism by which the driver fork routine, when resumed by a timeout, tests the timeout bit in the UCB and branches, if the bit is set, to the address of the timeout code.
- You must replace direct control and status register (CSR) access (for instance, by means of a MOVL instruction) with CSR access using one of the OpenVMS Alpha CSR access methods (CRAMs, platform independent access routines, or direct mapping).

Timeout Handling Code (Kernel Process)

Takes whatever action is necessary when a device has not yet responded to a request for device activity, and the time allowed for a response has expired.

Format

BLBC timeout-code-address

Arguments

None.

Essentials

Identifying the Routine

Specify the address of the timeout code in the **excpt** argument to the KP_STALL_WFIKPCH or KP_STALL_WFIRLCH macro.

Branched to

The KP_STALL_WFIKPCH, and KP_STALL_WFIRLCH macros use this entry point, but only when the label of timeout code is provided in their **excpt** argument. These macros are used in the driver's start-I/O routine; thus, strictly speaking, the driver itself is the only entity that uses this entry point.

The OpenVMS Alpha software timer interrupt service routine restarts a stalled driver kernel process fork procedure, passing a status (UCB$V_TIMOUT in UCB$L_STS) to it, which is inspected by one of two instructions left at the top of the fork procedure by the KP_STALL_WFIKPCH or KP_STALL_WFIRLCH macro. If UCB$V_TIMOUT is set, the second instruction branches to the timeout code.

Context

The timeout code receives control at device IPL and must exit at device IPL. At the time the timeout code executes, the processor holds both the fork lock and device lock associated with the device.

After taking whatever device-specific action is necessary at device IPL, timeout code can lower IPL to fork IPL to perform less critical activities. Because its caller restores IPL to fork IPL (and releases the device lock), if timeout code lowers IPL, it can do so only by forking or by performing the following steps:

1. Issue a DEVICEUNLOCK macro to lower to fork level
2. Perform timeout handling activities possible at the lower IPL
3. Issue a DEVICELOCK macro to again obtain the device lock and raise to device IPL

Timeout code can access only those virtual addresses that refer to system (S0) space.

Kernel process timeout code executes in the context of the kernel process thread that invoked the KP_STALL_WFIKPCH or KP_STALL_WFIRLCH macro.

Exit mechanism

Kernel process timeout code executes as part of the kernel process thread that invoked WFIKPCH or WFIRLCH macro and therefore has no special exit mechanism.

Description

There are no outputs required from timeout code but, depending on the characteristics of the device, timeout code might cancel or retry the current I/O request, send a message to the operator, or take some other action.

Before timeout code executes, OpenVMS has placed the device in a state in which no interrupt is expected (by clearing the bit UCB$V_INT in field UCB$L_STS). If the requested interrupt occurs while this routine executes, it will appear to be an unsolicited interrupt. Many drivers handle this situation by disabling interrupts while timeout code executes.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- On OpenVMS VAX systems, the software timer interrupt service routine issues a JSB instruction to a timeout handling routine within a driver when it detects that a device has timed out. On OpenVMS Alpha systems, the OpenVMS Alpha suspension macros provide a mechanism by which the driver fork routine, when resumed by a timeout, tests the timeout bit in the UCB and branches, if the bit is set, to the address of the timeout code.
- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access using one of the OpenVMS Alpha CSR access methods (CRAMs, platform independent access routines, or direct mapping).

Unit Delivery Routine

For controllers that can control a variable number of device units, determines which specific devices are present and available for inclusion in the system's configuration.

Format

status = DELIVER (ddb, idb, unit_number, scratch_area, adp)

Arguments

Argument	Type	Access	Mechanism
ddb	DDB	input	reference
idb	IDB	input	reference
unit_number	integer	input	value
scratch_area	address	input	reference
adp	ADP	input	reference

ddb
Device data block.

idb
Interrupt dispatch block; 0 if none exists.

unit_number
Number of unit that the unit delivery routine must decide to configure or not to configure.

scratch_area
Address of quadword scratch area.

adp
Adapter control block.

Essentials

Identifying the Routine
Specify the name of the unit delivery routine in the **deliver** argument to the DPTAB macro. The macro puts the procedure value address of this routine in the DPT.

Declaring the Entry Point
Use:

```
$DRIVER_DELIVER_ENTRY [preserve=<R2>] [,fetch=YES]
```

where:

> **preserve** indicates the registers to be preserved (in addition to those automatically preserved by the MACRO compiler) across the call to the unit delivery routine.
> **fetch**=**YES**, the default, loads the address of the IDB into R3 and R4, the unit number into R5, the address of the scratch area into R7, and the address of the ADP into R8; **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver unit delivery routine that uses this macro can access any of its arguments by using a symbolic name of the form DLVRARG$_**argument-name**.

Called by

The System Management (SYSMAN) utility's IO AUTOCONFIGURE command calls the unit delivery routine once for each unit the controller is capable of controlling. This value is specified in the **defunits** argument to the DPTAB macro.

Context

The unit delivery routine is called at IPL$_POWER. It must not lower IPL. The unit delivery routine executes in the context of the process within which the autoconfiguration facility executes.

Exit mechanism

A unit delivery routine returns success or failure status to the autoconfiguration facility. If the routine returns error status, the unit is not configured.

Description

The unit delivery routine determines which units on a controller should be configured. For instance, a unit delivery routine can prevent the creation of UCBs for devices that do not respond to a test for their presence.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- You must indicate the entry point of the routine with a $DRIVER_DELIVER_ENTRY macro to indicate which registers must be saved and restored across routine execution.
- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access using one of the OpenVMS Alpha CSR access methods (CRAMs, platform independent access routines, or direct mapping).
- The unit delivery routine of an OpenVMS VAX device driver receives the addresses of the device CSR in R4 and the IDB in R5. An OpenVMS Alpha device driver's unit delivery routine is not passed the address of the CSR. It may access the controller's register by means of the controller register access mailbox (CRAM), the address of which is provided in IDB$PS_CRAM.
- An OpenVMS Alpha unit delivery routine is passed the address of the device data block and the address of a quadword scratch area.

Unit Initialization Routine

Prepares a device for operation and, in the case of a device on a dedicated controller, initializes the controller.

Format

status = UNITINIT (idb, ucb)

Arguments

Argument	Type	Access	Mechanism
idb	IDB	input	reference
ucb	UCB	input	reference

idb

Interrupt dispatch block associated with the controller.

ucb

Unit control block.

Essentials

Identifying the Routine

Specify the address of the unit initialization routine **unitinit** argument of the DDTAB macro. This macro places the procedure value of the routine into the DDT.

Declaring the Entry Point

Use:

$DRIVER_UNITINIT_ENTRY [preserve=<R2>] [,fetch=YES]

where:

> **preserve** indicates the registers to be preserved (in addition to those automatically preserved by the MACRO-32 compiler) across the call to the unit initialization routine.
> **fetch**=**YES**, the default, loads $SS_NORMAL status into R0, and the addresses of the IDB and UCB into R4 and R5, respectively; **fetch**=**NO** disables register loading. Regardless of the value of the **fetch** argument, a driver unit initialization routine that uses this macro can access any of its arguments by using a symbolic name of the form UNITARG$_**argument-name**.

Called by

The driver-loading procedure calls a driver's unit initialization routine when processing a CONNECT command. OpenVMS calls a unit initialization routine when the device, the controller, the processor, or the adapter to which the device is connected undergoes power failure recovery.

Context

OpenVMS calls a unit initialization routine at IPL$_POWER. If it must lower IPL, the controller initialization routine cannot explicitly do so. Rather, it must fork. Because the driver-loading procedure calls the unit initialization routine immediately after the controller initialization returns control to it, the driver's initialization routines must synchronize their activities.

The portion of the unit initialization routine that services power failure cannot acquire any spin locks. As a result, the routine cannot fork to perform power failure servicing.

Because OpenVMS calls it in system context, a unit initialization routine can only refer to those virtual addresses that reside in system (S0) space. R0, and preserve the contents of all registers except R0, R1, and R2.

Exit mechanism

A unit initialization routine returns success or failure status to its caller.

Description

Depending on the device, a unit initialization routine performs any or all of the following tasks:

1. Determines whether it is being called as a result of a power failure by examining the power bit (UCB$V_POWER in UCB$L_STS) in the UCB. A unit initialization routine may want to perform or avoid specific tasks when servicing a power failure.
2. Clears error-status bits in device registers.
3. Enables controller interrupts.
4. Sets the online bit (UCB$V_ONLINE in UCB$L_STS).
5. Allocates resources that must be permanently allocated to the device or, for some devices, the controller.
6. If the device has a dedicated controller, as some printers do, fills in IDB$PS_OWNER.
7. For dedicated controllers, initializes controller and device hardware.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- You must indicate the entry point of the routine with a .JSB_ENTRY MACRO-32 compiler directive to indicate which registers are provided as input or used as output and which must be saved and restored.
- An OpenVMS VAX device driver can specify a controller initialization routine by invoking the DPT_STORE macro to place its address into the interrupt transfer vector block (CRB$L_INTD+VEC$L_UNITINIT). An OpenVMS Alpha device driver specifies the routine in the **unitinit** argument of the DDTAB macro.
- You must replace direct CSR access (for instance, by means of a MOVL instruction) with CSR access using one of the OpenVMS Alpha CSR access methods (CRAMs, platform independent access routines, or direct mapping).

- The unit initialization routine of an OpenVMS VAX device driver receives the addresses of the primary and secondary device CSRs in R3 and R4, respectively. An OpenVMS Alpha device driver's unit initialization routine is not passed the addresses of the CSRs. It may access the controller registers by means of the controller register access mailbox (CRAM), the address of which is provided in IDB$PS_CRAM.
- An OpenVMS Alpha unit initialization routine must return success or failure status to its caller.

9

System Routines

This chapter describes the operating system routines that are used by device drivers and employs the following conventions:

- Most routines reside in modules within the [SYS] facility of the operating system. A routine description provides a facility name (in brackets) only if the module is not located in the [SYS] facility.
- Many routines are not directly called by device drivers. Rather, the operating system supplies macros that drivers invoke to accomplish the routine call. The description of a routine that has such a macro interface lists the name of the associated macro. Chapter 11 describes how a driver can use these macros.
- System routines generally return a status value in R0 (for instance, SS$_NORMAL). The low-order bit of this value indicates successful (1) or unsuccessful (0) completion of the routine. Additional information on returned status values appears in the *OpenVMS System Services Reference Manual* and the *OpenVMS System Messages and Recovery Procedures Reference Manual*.

Table 9–1 highlights some of the differences between OpenVMS VAX and OpenVMS Alpha system routines.

Table 9–1 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

(continued on next page)

Table 9–1 (Cont.) New, Changed, and Unsupported OpenVMS System Routines

System Routine	Description	Notes
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$KP_ALLOCATE_KPB	Creates a KPB and a kernel process stack, as required by the kernel process services	New
EXE$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

(continued on next page)

Table 9–1 (Cont.) New, Changed, and Unsupported OpenVMS System Routines

System Routine	Description	Notes
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
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

(continued on next page)

Table 9–1 (Cont.) New, Changed, and Unsupported OpenVMS System Routines

System Routine	Description	Notes
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$DEALLOCATE_CRAM	Deallocates a controller register access mailbox	New
IOC$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

(continued on next page)

Table 9–1 (Cont.) New, Changed, and Unsupported OpenVMS System Routines

System Routine	Description	Notes
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
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

ACP_STD$ACCESS

Accesses and creates ACP function processing.

Module

SYSACPFDT

Format

status = ACP_STD$ACCESS (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_DEVNOTMOUNT	Device not mounted.
SS$_DEVFOREIGN	Device is mounted as foreign.
SS$_EXQUOTA	File quota exceeded.
SS$_FILALRACC	File already accessed.

SS$_IVCHNLSEC	Invalid section channel.
SS$_NORMAL	The I/O request has been successfully queued to the appropriate ACP or XQP.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$ACCESS as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine ACP$ACCESS expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$ACCESS.
- ACP$ACCESS returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$ACCESS returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$ACCESSNET

Connects to network function processing.

Module

SYSACPFDT

Format

status = ACP_STD$ACCESSNET (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_NORMAL	The I/O request has been successfully queued to the appropriate ACP or XQP.
SS$_EXQUOTA	File quota exceeded.
SS$_FILALRACC	File already accessed.
SS$_IVCHNLSEC	Invalid section channel.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$ACCESSNET as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine ACP$ACCESSNET (used by OpenVMS VAX drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$ACCESSNET.
- ACP$ACCESSNET returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$ACCESSNET returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$DEACCESS

Deaccesses ACP function processing.

Module

SYSACPFDT

Format

status = ACP_STD$DEACCESS (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_FILNOTACC	File not accessed.
SS$_IVCHNLSEC	Invalid section channel.
SS$_NORMAL	Normal, successful completion.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$DEACCESS as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine ACP$DEACCESS (used by OpenVMS VAX drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$DEACCESS.
- ACP$DEACCESS returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$DEACCESS returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$MODIFY

Deletes and modifies ACP function processing.

Module

SYSACPFDT

Format

status = ACP_STD$MODIFY (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_DEVNOTMOUNT	Device not mounted.
SS$_DEVFOREIGN	Device is mounted as foreign.
SS$_EXQUOTA	File quota exceeded.
SS$_NORMAL	The I/O request has been successfully queued to the appropriate ACP or XQP.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$MODIFY as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine ACP$MODIFY (used by OpenVMS VAX drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$MODIFY.
- ACP$MODIFY returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$MODIFY returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$MOUNT

Initiates ACP mount function processing.

Module

SYSACPFDT

Format

status = ACP_STD$MOUNT (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_DEVNOTMOUNT	Device not mounted.
SS$_NOPRIV	Process has insufficient privileges.
SS$_NORMAL	The I/O request has been successfully queued to the appropriate ACP or XQP.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$MOUNT as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine ACP$MOUNT (used by OpenVMS VAX drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$MOUNT.
- ACP$MOUNT returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$MOUNT returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$READBLK

Processes a read block ACP function.

Module

SYSACPFDT

Format

status = ACP_STD$READBLK (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_ENDOFFILE	End of file reached.
SS$_FILNOTACC	File not accessed on channel.
SS$_NOPRIV	Process has insufficient privileges.
SS$_ILLIOFUNC	Illegal I/O function.

SS$_ILLBLKNUM	Illegal block number.
SS$_NORMAL	Normal, successful completion.
SS$_INSFWSL	Insufficient working set limit.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$READBLK as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine ACP$READBLK (used by OpenVMS VAX drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$READBLK.
- ACP$READBLK returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$READBLK returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

ACP_STD$WRITEBLK

Processes a write block ACP function.

Module

SYSACPFDT

Format

status = ACP_STD$WRITEBLK (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Access violation.
SS$_BADPARAM	Record size is too small for magtape function processing.
SS$_ENDOFFILE	End of file reached.
SS$_FILNOTACC	File not accessed on channel.
SS$_NOPRIV	Process has insufficient privileges.

SS$_ILLIOFUNC	Illegal I/O function.
SS$_ILLBLKNUM	Illegal block number.
SS$_INSFMEM	Insufficient memory to perform erase function.
SS$_INSFSPTS	Insufficient system page table entries to perform erase function.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Normal, successful completion.
SS$_WRITLCK	Device software is write locked.

Context

FDT dispatching code in the $QIO system service calls ACP_STD$WRITEBLK as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine ACP$WRITEBLK (used by OpenVMS VAX drivers) expects, as input, a bit number indicating the requested I/O function in R7, and the address of the FDT entry from which it received control in R8.

  R0, R7, and R8 are not provided as input to ACP_STD$WRITEBLK.
- ACP$WRITEBLK returns control to the system service dispatcher, passing it the final $QIO system service status in R0. ACP_STD$WRITEBLK returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

COM_STD$DELATTNAST

Delivers all attention ASTs linked in the specified list.

Module

COMDRVSUB

Format

COM_STD$DELATTNAST (acb_lh, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
acb_lh	address	input	reference	required
ucb	UCB	input	reference	required

ast_lh
Listhead of AST control blocks

ucb
Unit control block.

Context

COM_STD$DELATTNAST executes and exits at the caller's IPL, and acquires no spin locks. However, the caller must be executing at IPL$_RESCHED or higher to avoid certain race conditions.

Description

COM_STD$DELATTNAST removes all AST control blocks (ACBs) from the specified list. Using each ACB as a fork block, it schedules a fork process at IPL$_QUEUEAST to queue the AST to its target process. COM_STD$DELATTNAST dequeues each ACB from the head of the list, thus removing them in the reverse order of their declaration by COM_STD$SETATTNAST. Note that in certain circumstances attention ASTs can be delivered to a user process before the delivery of I/O completion ASTs previously posted by the driver.

Macro

CALL_DELATTNAST [save_r0r1]

where:

> **save_r0r1** indicates that the macro should preserve registers R0 and R1 across the call to COM_STD$DELATTNAST. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

In an Alpha driver, CALL_DELATTNAST calls COM_STD$DELATTNAST using the current contents of R4 and R5 as the **listhead** and **ucb** arguments, respectively. Unless you specify **save_r0r1**=**NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note that COM_STD$DELATTNAST replaces COM$DELATTNAST. Unlike COM$DELATTNAST, COM_STD$DELATTNAST does not preserve the contents of R0 and R1.

COM_STD$DELATTNASTP

Delivers all attention ASTs linked in the specified list for a given process.

Module

COMDRVSUB

Format

COM_STD$DELATTNASTP (acb_lh, ucb, ipid)

Arguments

Argument	Type	Access	Mechanism	Status
acb_lh	listhead	input	reference	required
ucb	UCB	input	reference	required
ipid	integer	input	value	required

acb_lh
Listhead of AST control blocks

ucb
Unit control block.

ipid
Internal process ID (IPID) for the target process.

Context

COM_STD$DELATTNASTP executes and exits at the caller's IPL, and acquires no spin locks. However, the caller must be executing at IPL$_RESCHED or higher to avoid certain race conditions.

Description

For Digital internal use only.

Macro

CALL_DELATTNASTP [save_r0r1]

where:

> **save_r0r1** indicates that the macro should preserve registers R0 and R1 across the call to COM_STD$DELATTNASTP. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

In an Alpha driver, CALL_DELATTNASTP calls COM_STD$DELATTNASTP using the current contents of R4, R5 and R6 as the **listhead**, **ucb**, and **ipid** arguments, respectively. Unless you specify **save_r0r1**=**NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- COM_STD$DELATTNASTP replaces COM$DELATTNASTP. Unlike COM$DELATTNASTP, COM_STD$DELATTNASTP does not preserve the contents of R0 and R1.

COM_STD$DELCTRLAST

Delivers all control ASTs, linked in the specified list, that match a given condition.

Module

COMDRVSUB

Format

COM_STD$DELCTRLAST (acb_lh, ucb, matchchar, inclchar_p)

Arguments

Argument	Type	Access	Mechanism	Status
acb_lh	listhead	input	reference	required
ucb	UCB	input	reference	required
matchchar	integer	input	value	required
inclchar_p	pointer	output	value	required

acb_lh
Listhead of AST control blocks

ucb
Unit control block.

matchchar
Match character.

inclchar_p
Address in which COM_STD$DELCTRLAST writes the character to include in the data stream, or NULL.

Context

COM_STD$DELCTRLAST executes and exits at the caller's IPL, and acquires no spin locks. However, the caller must be executing at IPL$_RESCHED or higher to avoid certain race conditions.

Description

For Digital internal use only.

Macro

CALL_DELCTRLAST [save_r0r1]

where:

> **save_r0r1** indicates that the macro should preserve registers R0 and R1 across the call to COM_STD$DELCTRLAST. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

In an Alpha driver, CALL_DELCTRLAST calls COM_STD$DELCTRLAST using the current contents of R4, R5, and R3 as the **listhead**, **ucb**, and **matchchar** arguments, respectively. When COM$DELCTRLAST returns, it moves the include character into R3. Unless you specify **save_r0r1**=**NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- COM_STD$DELCTRLAST replaces COM$DELCTRLAST. Unlike COM$DELCTRLAST, COM_STD$DELCTRLAST does not preserve the contents of R0 and R1.

COM_STD$DELCTRLASTP

Delivers all control ASTs, linked in the specified list, that match a given condition.

Module

COMDRVSUB

Format

COM_STD$DELCTRLASTP (acb_lh, ucb, ipid, matchchar, inclchar_p)

Arguments

Argument	Type	Access	Mechanism	Status
acb_lh	listhead	input	reference	required
ucb	UCB	input	reference	required
ipid	integer	input	value	required
matchchar	integer	input	value	required
inclchar_p	pointer	input	value	required

acb_lh
Listhead of AST control blocks

ucb
Unit control block.

ipid
Internal process ID (IPID) for the target process.

matchchar
Match character.

inclchar_p
Address in which COM_STD$DELCTRLAST writes the character to include in the data stream, or NULL.

Context

COM_STD$DELCTRLASTP executes and exits at the caller's IPL, and acquires no spin locks. However, the caller must be executing at IPL$_RESCHED or higher to avoid certain race conditions.

Description

For Digital internal use only.

Macro

CALL_DELCTRLASTP [save_r0r1]

where:

> **save_r0r1** indicates that the macro should preserve registers R0 and R1 across the call to COM_STD$DELCTRLASTP. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

In an Alpha driver, CALL_DELCTRLASTP calls COM_STD$DELCTRLASTP using the current contents of R4, R5, R6, and R3 as the **listhead**, **ucb**, **ipid**, and **matchchar** arguments, respectively. When COM$DELCTRLASTP returns, it moves the include character into R3. Unless you specify **save_r0r1**=**NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- COM_STD$DELCTRLASTP replaces COM$DELCTRLASTP. Unlike COM$DELCTRLASTP, COM_STD$DELCTRLASTP does not preserve the contents of R0 and R1.

COM_STD$DRVDEALMEM

Deallocates system dynamic memory.

Module

MEMORYALC_MIN or MEMORYALC_MON

Format

COM_STD$DRVDEALMEM (block)

Arguments

Argument	Type	Access	Mechanism	Status
ptr	structure	input	reference	required

ptr
Block to be deallocated. The block must be a standard OpenVMS data structure (in which offset FKB$W_SIZE contains its size). The block size must be at least FKB$K_LENGTH (24 bytes). (The FKB$ symbols are defined by the $FKBDEF macro in SYS$LIBRARY:LIB.MLB.)

Context

A driver can call COM_STD$DRVDEALMEM from any IPL. COM_STD$DRVDEALMEM executes at the caller's IPL and returns control at that IPL. The caller retains any spin locks it held at the time of the call.

Description

COM_STD$DRVDEALMEM transfers control to EXE$DEANONPAGED to deallocate the buffer specified by the **block** parameter. If COM_STD$DRVDEALMEM cannot deallocate memory at the caller's IPL, it transforms the block being deallocated into a fork block and queues the block in the fork queue. The code that executes in the fork process then jumps to EXE$DEANONPAGED.

If the buffer to be deallocated is less than FKB$C_LENGTH in size, or its address is not aligned on a 16-byte boundary, COM_STD$DRVDEALMEM issues a BADDALRQSZ bugcheck.

Macro

CALL_DRVDEALMEM [save_r0r1]

where:

save_r0r1 indicates that the macro should preserve registers R0 and R1 across the call to COM_STD$DRVDEALMEM. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

In an Alpha driver, CALL_DRVDEALMEM calls COM_STD$DRVDEALMEM using the current contents of R0 as the **ptr** argument. Unless you specify **save_r0r1**=**NO**, the macro preserves the quadword registers R0 and R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- COM_STD$DRVDEALMEM replaces COM$DRVDEALMEM (OpenVMS VAX drivers). Unlike COM$DRVDEALMEM, COM_STD$DRVDEALMEM does not preserve the contents of R0 and R1.

COM_STD$FLUSHATTNS

Removes specified ASTs from an attention AST list.

Module

COMDRVSUB

Format

status = COM_STD$FLUSHATTNS (pcb, ucb, chan ,acb_lh)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
chan	integer	input	value	required
acb_lh	listhead	input	reference	required

pcb

Process control block. COM_STD$FLUSHATTNS reads the following PCB fields:

Field	Contents
PCB$L_PID	Process ID
PCB$L_ASTCNT	ASTs remaining in quota

COM_STD$FLUSHATTNS increases PCB$L_ASTCNT once for each AST control block (ACB) it flushes.

ucb

Unit control block. COM_STD$FLUSHATTNS reads UCB$L_DLCK to obtain the address of the device lock.

chan

Number of the assigned I/O channel.

acb_lh

Listhead of ACBs.

Return Values

SS$_NORMAL	Normal, successful completion

Context

COM_STD$FLUSHATTNS raises IPL to device IPL, acquiring the corresponding device lock. Before returning control to its caller at the caller's IPL, COM_STD$FLUSHATTNS releases the device lock. The caller retains any spin locks it held at the time of the call.

Description

A driver's cancel-I/O routine calls COM_STD$FLUSHATTNS to flush an attention AST list. A driver FDT routine calls COM_STD$FLUSHATTNS to service a $QIO request that specifies a set-attention-AST function and a value of 0 in the **p1** argument (IRP$L_QIO_P1).

COM_STD$FLUSHATTNS locates all ACBs blocks whose channel number and PID match those supplied as input to the routine. It removes them from the specified list, deallocates them, and returns control to its caller.

Macro

CALL_FLUSHATTNS

In an Alpha driver, CALL_FLUSHATTNS calls COM_STD$FLUSHATTNS using the current contents of R4, R5, R6, and R7 as the **pcb**, **ucb**, **chan**, and **acb_lh** arguments, respectively. It returns status in R0.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- COM_STD$FLUSHATTNS replaces COM$FLUSHATTNS (used by OpenVMS VAX drivers).

COM_STD$FLUSHCTRLS

Removes specified ASTs from a control AST list.

Module

COMDRVSUB

Format

status = COM_STD$FLUSHCTRLS (pcb, ucb, chan ,acb_lh, mask_p)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
chan	integer	input	value	required
acb_lh	listhead	input	reference	required
mask_p	mask_ longword	input	reference	required

pcb
Process control block. COM_STD$FLUSHCTRLS reads the following PCB fields:

Field	Contents
PCB$L_PID	Process ID
PCB$L_ASTCNT	ASTs remaining in quota

COM_STD$FLUSHCTRLS increases PCB$L_ASTCNT once for each control AST control block (TAST) it flushes.

ucb
Unit control block. COM_STD$FLUSHCTRLS reads UCB$L_DLCK to obtain the address of the device lock.

chan
Number of the assigned I/O channel.

acb_lh
Listhead of ACBs.

mask_p
Summary mask of active control characters. COM_STD$FLUSHCTRLS updates this mask.

Return Values

SS$_NORMAL	Normal, successful completion

Context

COM_STD$FLUSHCTRLS raises IPL to device IPL, acquiring the corresponding device lock. Before returning control to its caller at the caller's IPL, COM_STD$FLUSHCTRLS releases the device lock. The caller retains any spin locks it held at the time of the call.

Description

For Digital internal use only.

Macro

CALL_FLUSHCTRLS

In an Alpha driver, CALL_FLUSHCTRLS calls COM_STD$FLUSHCTRLS using the current contents of R2, R4, R5, R6, and R7 as the **mask**, **pcb**, **ucb**, **chan**, and **acb_lh** arguments, respectively. It returns status in R0.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- COM_STD$FLUSHCTRLS replaces COM$FLUSHCTRLS (used by OpenVMS VAX drivers).

COM_STD$POST, COM_STD$POST_NOCNT

Initiate device-independent postprocessing of an I/O request independent of the status of the device unit.

Module

COMDRVSUB

Format

COM_STD$POST (irp, ucb)

COM_STD$POST_NOCNT (irp)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request block. The following IRP fields are input to I/O postprocessing.

Field	Contents
IRP$L_MEDIA	Data to be copied to the I/O status block
IRP$L_MEDIA+4	Data to be copied to the I/O status block

ucb
Unit control block (COM_STD$POST only). COM_STD$POST increases the unit operation count (UCB$L_OPCNT).

Context

Drivers call COM_STD$POST at or above fork IPL. Drivers call COM_STD$POST_NOCNT at or above IPL$_ASTDEL. These routines execute at their caller's IPL and return control at that IPL. The caller retains any spin locks it held at the time of the call.

Description

A driver fork process calls COM_STD$POST or COM_STD$POST_NOCNT after it has completed device-dependent I/O processing for an I/O request initiated by EXE_STD$ALTQUEPKT. Because COM_STD$POST_NOCNT, unlike COM_STD$POST, does not increment the unit's operations count (UCB$L_OPCNT), a driver uses COM_STD$POST_NOCNT to initiate completion processing for an I/O request when the associated UCB is not available.

COM_STD$POST and COM_STD$POST_NOCNT insert the IRP into the systemwide I/O postprocessing queue, request an IPL$_IOPOST software interrupt, and return control to the caller. Unlike IOC_STD$REQCOM, these routines do not attempt to dequeue any IRP waiting for the device or change the busy status of the device.

Macro

CALL_POST [save_r1]
CALL_POST_NOCNT [save_r1]

where:

> **save_r1** indicates that the macro should preserve register R1 across the call to COM_STD$POST or COM_STD$POST_NOCNT. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

In an Alpha driver, CALL_POST calls COM_STD$POST using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively. CALL_POST_NOCNT simulates a JSB to COM$POST_NOCNT. It calls COM_STD$POST_NOCNT using the current contents of R3 as the **irp** argument. Unless you specify **save_r1**=**NO**, the macro preserves the quadword register R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- COM_STD$POST replaces COM$POST (used by OpenVMS VAX drivers); COM_STD$POST_NOCNT replaces COM$POST_NOCNT. The Alpha routines do not preserve R1 across the call.

COM_STD$SETATTNAST

Enables or disables attention ASTs.

Module

COMDRVSUB

Format

status = COM_STD$SETATTNAST (irp, pcb, ucb, ccb, acb_lh)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required
acb_lh	listhead	input	reference	required

irp
I/O request packet for the current I/O request.

COM_STD$SETATTNAST reads the following IRP fields:

Field	Contents
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the address of the AST routine, or zero to flush the AST queue.
IRP$L_QIO_P2	$QIO system service **p2** argument, containing the AST parameter.
IRP$L_QIO_P3	$QIO system service **p3** argument, containing the access mode of the AST request.
IRP$L_CHAN	I/O request channel index number.

pcb
Process control block of the current process.

COM_STD$SETATTNAST reads the following PCB fields:

Field	Contents
PCB$L_ASTCNT	Number of ASTs remaining in process quota
PCB$L_PID	Process ID

COM_STD$SETATTNAST decreases PCB$L_ASTCNT if it successfully queues the AST.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

COM_STD$SETATTNAST reads UCB$L_DLCK.

ccb
Channel control block that describes the process-I/O channel

acb_lh
Address of listhead of AST control blocks.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.
SS$_NORMAL	Normal, successful completion

Status in FDT_CONTEXT

SS$_EXQUOTA	Process AST quota exceeded.
SS$_INSFMEM	No memory available to allocate the expanded ACB.

Context

The FDT support routine COM_STD$SETATTNAST must be called from code executing at IPL$_ASTDEL. COM_STD$SETATTNAST raises IPL and acquires the corresponding device lock, to insert the AST into the AST queue. It returns control to its caller at IPL$_ASTDEL.

Description

A driver FDT routine calls COM_STD$SETATTNAST to service a $QIO request that specifies a set-attention-AST function.

If the **p1** argument of the request contains a zero, COM_STD$SETATTNAST transfers control to COM_STD$FLUSHATTNS, which disables all ASTs indicated by the PID and I/O channel number (IRP$L_CHAN). COM_STD$FLUSHATTNS searches through the AST control block (ACB) list, extracts each identified ACB, deallocates it, and returns SS$_NORMAL status in R0 to COM_STD$SETATTNAST. COM_STD$SETATTNAST returns this status to its caller.

If the **p1** argument of the request contains the address of an AST routine, COM_STD$SETATTNAST decreases PCB$L_ASTCNT and allocates an expanded AST control block (ACB) that contains the following information:

- Spin lock index SPL$C_QUEUEAST
- Address of the AST routine (as specified in **p1**)
- AST parameter (as specified in **p2**)
- Access mode (the maximum, or least privileged, access mode between the access mode specified in **p3** and the current process's access mode). Bit ACB$V_QUOTA is set in this value to indicate that the AST was requested by a process, not by the system.

- Number of the assigned I/O channel
- PID of the requesting process

COM_STD$SETATTNAST links the ACB to the start of the specified linked list of ACBs located in a UCB extension area. COM$DELATTNAST can later use the expanded ACB to fork to IPL$_QUEUEAST, at which IPL it reformats the block into a standard ACB.

If the process exceeds its AST quota, or if there is no memory available to allocate the expanded ACB, COM_STD$SETATTNAST restores PCB$L_ASTCNT to its original value and calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, COM_STD$SETATTNAST returns to its caller with this status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.

The caller of COM_STD$SETATTNAST must examine the status in R0:

- If the status is SS$_NORMAL, the attention AST has been enabled (or the AST has been flushed), as requested.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the operation to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS.

Macro

CALL_SETATTNAST

In an Alpha driver, CALL_SETATTNAST calls COM_STD$SETATTNAST using the current contents of R3, R4, R5, R6, and R7, as the **irp**, **pcb**, **ucb**, **ccb**, and **acb_lh** arguments, respectively. It returns status in R0 and in the FDT_CONTEXT structure.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- COM_STD$SETATTNAST replaces COM$SETATTNAST. COM$SETATTNAST returns to its caller only upon success; COM_STD$SETATTNAST returns to its caller whether it has been successful or not. It returns SS$_NORMAL or SS$_FDT_COMPL status in R0. When it returns SS$_FDT_COMPL status, the FDT_CONTEXT structure contains additional status (SS$_EXQUOTA or SS$_INSFMEM) to explain why the request has been aborted.
- COM$SETATTNAST preserves the addresses of the IRP and UCB in R3 and R5 across the call. COM_STD$SETATTNAST does not.

COM_STD$SETCTRLAST

Enables or disables control ASTs.

Module

COMDRVSUB

Format

status = COM_STD$SETCTRLAST (irp, pcb, ucb, acb_lh, mask, tast_p)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
acb_lh	listhead	input	reference	required
mask	mask_ longword	input	value	required
tast_p	TAST	output	value	required

irp
I/O request packet for the current I/O request.

COM_STD$SETCTRLAST reads the following IRP fields:

Field	Contents
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the address of the AST routine to call when an out-of-band character is typed, or zero to flush the queue.
IRP$L_QIO_P2	$QIO system service **p2** argument, containing the address of the short-form terminator mask, indicating which out-of-band characters precipitate AST delivery. This address is passed as an AST parameter when the AST is delivered.
IRP$L_QIO_P3	$QIO system service **p3** argument, containing the access mode of the AST request.
IRP$L_CHAN	I/O request channel index number

pcb
Process control block of the current process.

COM_STD$SETCTRLAST reads the following PCB fields:

Field	Contents
PCB$L_ASTCNT	Number of ASTs remaining in process quota
PCB$L_PID	Process ID

COM_STD$SETCTRLAST decreases PCB$L_ASTCNT if it successfully queues the AST.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

COM_STD$SETCTRLAST reads UCB$L_DLCK.

acb_lh
Address of listhead of AST control blocks.

mask
Summary mask of active control characters. COM_STD$SETCTRLAST updates the summary mask to be the inclusive-OR of all masks in the control AST list.

tast_p
Address of the control AST block (TAST), returned as output from COM_STD$SETCTRLAST.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.
SS$_NORMAL	Normal, successful completion

Status in FDT_CONTEXT

SS$_ACCVIO	Specified **mask** is not addressable.
SS$_EXQUOTA	Process AST quota exceeded.
SS$_INSFMEM	No memory available to allocate the expanded ACB.

Context

The FDT support routine COM_STD$SETCTRLAST must be called from code executing at IPL$_ASTDEL. COM_STD$SETCTRLAST raises IPL and acquires the corresponding device lock, to insert the AST into the AST queue. It returns control to its caller at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

CALL_SETCTRLAST

In an Alpha driver, CALL_SETCTRLAST calls COM_STD$SETCTRLAST using the current contents of R3, R4, R5, R7, and R2, as the **irp**, **pcb**, **ucb**, **acb_lh**, and **mask** arguments, respectively. It returns the TAST block in R2. It returns status in R0 and in the FDT_CONTEXT structure.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- COM_STD$SETCTRLAST replaces COM$SETCTRLAST. The order in which formal parameters are passed to COM_STD$SETCTRLAST differs from the order in which they are provided in registers to the COM$SETCTRLAST routine.
- COM_STD$SETCTRLAST does not provide the address of the TAST block as output in R2.
- COM$SETCTRLAST returns to its caller only upon success; COM_STD$SETCTRLAST returns to its caller whether it has been successful or not. It returns SS$_NORMAL or SS$_FDT_COMPL status in R0. When it returns SS$_FDT_COMPL status, the FDT_CONTEXT structure contains additional status (SS$_EXQUOTA or SS$_INSFMEM) to explain why the request has been aborted.

ERL_STD$ALLOCEMB

Allocates an error log message buffer and initializes its header.

Module

ERRORLOG

Format

status = ERL_STD$ALLOCEMB (size, embdv_p)

Arguments

Argument	Type	Access	Mechanism	Status
size	integer	input	value	required
embdv_p	pointer	output	reference	required

size
Size of requested error message buffer in bytes.

embdv_p
Address of a pointer in which ERL_STD$ALLOCEMB writes the address of the error message buffer (EMBDV).

Return Values

status	Low bit set indicates success, low bit clear indicates failure

Context

A driver can call ERL_STD$ALLOCEMB from any IPL. ERL_STD$ALLOCEMB raises IPL to IPL$_EMB and obtains the corresponding spin lock to allocate the error message buffer. It returns control to its caller at its caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

For Digital internal use only.

Macro

CALL_ALLOCEMB

In an Alpha driver, CALL_ALLOCEMB calls ERL_STD$ALLOCEMB using the current contents of R1 as the **size** argument. It returns status in R0, the address of the allocated EMB in R2 and copies the error log sequence number from EMB$W_DV_ERRSEQ to R1.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- ERL_STD$ALLOCEMB replaces ERL$ALLOCEMB. Unlike ERL$ALLOCEMB, ERL_STD$ALLOCEMB does not return the error sequence number in R1. A driver can obtain the error sequence number from the error message buffer (EMB$W_DV_ERRSEQ).

ERL_STD$DEVICEATTN, ERL_STD$DEVICERR, ERL_STD$DEVICTMO

Allocate an error message buffer and record in it information concerning the error.

Module

ERRORLOG

Format

ERL_STD$DEVICEATTN (driver_param, ucb)

ERL_STD$DEVICERR (driver_param, ucb)

ERL_STD$DEVICTMO (driver_param, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
driver_param	undefined	input	reference	required
ucb	UCB	input	reference	required

driver_param
Parameter to be passed to the register dumping routine, usually a controller register access mailbox (CRAM).

ucb
Unit control block. These routines read the following UCB fields:

Field	Contents
UCB$L_DEVCHAR	Bit DEV$V_ELG set.
UCB$L_FUNC	Bit IO$V_INHERLOG clear.
UCB$L_IRP	Address of IRP currently being processed (ERL_STD$DEVICERR and ERL_STD$DEVICTMO only).
UCB$L_ORB	ORB address.
UCB$L_DDB	DDB address.
UCB$L_DDT	DDT address. DDT$W_ERRORBUF contains the size of the error message buffer in bytes.

These routines write the following UCB fields:

Field	Contents
UCB$L_ERRCNT	Increased.
UCB$L_EMB	Address of error message buffer.
UCB$L_STS	UCB$V_ERLOGIP set.

Context

A driver calls ERL_STD$DEVICEATTN, ERL_STD$DEVICERR, or ERL_STD$DEVICTMO at or above fork IPL, holding the corresponding fork lock in an OpenVMS multiprocessing environment.

These routines return control to the caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

ERL_STD$DEVICERR and ERL_STD$DEVICTMO log an error associated with a particular I/O request. ERL_STD$DEVICEATTN logs an error that is not associated with an I/O request. Each of these routines performs the following steps:

1. Increases UCB$L_ERRCNT to record a device error. If the error-log-in-progress bit (UCB$V_ERLOGIP in UCB$L_STS) is set, the routine returns control to its caller.
2. Allocates from the current error log allocation buffer an error message buffer of the length specified in the device's DDT (in argument **erlgbf** to the DDTAB macro). This allocation is performed at IPL$_EMB holding the EMB spin lock.
3. Places the address of the error message buffer in UCB$L_EMB.
4. Sets UCB$V_ERLOGIP in UCB$L_STS.
5. Initializes the buffer with the current system time, error log sequence number, and error type code. These routines use the following error type codes:

ERL_STD$DEVICEATTN	Device attention (EMB$C_DA)
ERL_STD$DEVICERR	Device error (EMB$C_DE)
ERL_STD$DEVICTMO	Device timeout (EMB$C_DT)

6. Loads fields from the UCB, the IRP, and the DDB into the buffer, including the following:

UCB$B_DEVCLASS	Device class
UCB$B_DEVTYPE	Device type
IRP$L_PID	Process ID of the process originating the I/O request (ERL_STD$DEVICERR or ERL_STD$DEVICTMO)
IRP$L_BOFF	Transfer parameter (ERL_STD$DEVICERR and ERL_STD$DEVICTMO)
IRP$L_BCNT	Transfer parameter (ERL_STD$DEVICERR and ERL_STD$DEVICTMO)
IRP$L_MEDIA	Disk address
UCB$W_UNIT	Unit number
UCB$L_ERRCNT	Count of device errors
UCB$L_OPCNT	Count of completed operations
ORB$L_OWNER	UIC of volume owner

UCB$L_DEVCHAR	Device characteristics
IRP$L_FUNC	I/O function value (ERL_STD$DEVICERR and ERL_STD$DEVICTMO)
DDB$T_NAME	Device name (concatenated with cluster node name if appropriate)

7. Loads into R0 the address of the location in the buffer in which the contents of the device registers are to be stored.
8. Calls the driver's register dumping routine, the address of which is specified in the **regdmp** argument to the DDTAB macro.

Macro

CALL_DEVICEATTN [save_r0r1]
CALL_DEVICERR [save_r0r1]
CALL_DEVICTMO [save_r0r1]

where:

> **save_r0r1** indicates that the macros must preserve the contents of R0 and R1 across the call to ERL_STD$DEVICEATTN, ERL_STD$DEVICERR, or ERL_STD$DEVICTMO. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

In an Alpha driver, the CALL_DEVICEATTN, CALL_DEVICERR, and CALL_DEVICTMO macros call corresponding routines using the current contents of R4 and R5 as the **driverpar** and **ucb** arguments, respectively. Unless you specify **save_r0r1**=**NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- ERL_STD$DEVICEATTN, ERL_STD$DEVICERR, and ERL_STD$DEVICTMO replace ERL$DEVICEATTN, ERL$DEVICERR, and ERL$DEVICTMO. The Alpha routines do not preserve the contents of R0 and R1.
- Because the UCB$L_MEDIA field has been removed from the UCB local disk extension, these routines write the disk address into the EMB from IRP$L_MEDIA.
- Because the UCB$B_SLAVE field has been removed from the UCB local disk extension, these routines do not write that field.
- OpenVMS Alpha device drivers consequently do not need to define the local disk UCB extension or local tape UCB extension to use these error logging routines.
- **driver_param** is considered required input to these routines.

ERL_STD$RELEASEMB

Releases an error message buffer to the error logging process.

Module

ERRORLOG

Format

ERL_STD$RELEASEMB (embdv)

Arguments

Argument	Type	Access	Mechanism	Status
embdv	EMBDV	input	reference	required

embdv
Error message buffer to be released.

Context

A driver can call ERL_STD$RELEASEMB from any IPL. ERL_STD$RELEASEMB raises IPL to IPL$_EMB and obtains the corresponding spin lock to release the error message buffer. It returns control to its caller at its caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

For Digital internal use only.

Macro

CALL_RELEASEMB

In an Alpha driver, CALL_RELEASEMB calls ERL_STD$RELEASEMB using the current contents of R2 as the **buff** argument.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- ERL_STD$RELEASEMB replaces ERL$RELEASEMB.

EXE$BUS_DELAY

Allows a system-specific bus delay within a timed wait.

Module

[.SYSLOA]TIMEDWAIT

Macro

TIMEDWAIT

Format

EXE$BUS_DELAY adp

Context

EXE$BUS_DELAY conforms to the OpenVMS calling standard.

Arguments

adp

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Address of ADP.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_INSFARG	Not all of the required arguments were specified.

Description

The OpenVMS VAX version of the TIMEDWAIT macro generated a processor-specific delay for the bus indicated by the ADP before executing the series of instructions, specified in the macro invocation, that check for the occurrence of a specific event or condition. In OpenVMS VAX systems, the delay helps prevent flooding the bus paths with references to device interface registers in I/O space.

An implicit call to EXE$BUS_DELAY is included in the expansion of the TIMEDWAIT macro when you specify the **bus** argument. You can explicitly call EXE$BUS_DELAY but, if you do, you must not also employ the TIMEDWAIT macro with the **bus** argument.

Note

In OpenVMS Alpha, EXE$BUS_DELAY checks for the required argument and, if it is present, returns to its caller with SS$_NORMAL status.

EXE$DELAY

Provides a short-term simple delay.

Module

[SYSLOA]TIMEDWAIT

Macro

TIMEDELAY

Format

EXE$DELAY delta

Context

EXE$DELAY conforms to the OpenVMS calling standard.

Arguments

delta

VMS Usage:	aligned quadword
type:	quadword (unsigned)
access:	read only
mechanism:	by reference

Delay time specified in nanoseconds.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_INSFARG	Not all of the required arguments were specified.

Description

EXE$DELAY implements a simple delay by looping for at least the requested time interval. System events such as interrupt processing may have some impact on the actual time delay.

EXE$KP_ALLOCATE_KPB

Creates a KPB and a kernel process stack, as required by the OpenVMS kernel process services.

Module

KERNEL_PROCESS_MIN, KERNEL_PROCESS_MON

Macro

KP_ALLOCATE_KPB
DDTAB (**start**=EXE$KP_STARTIO)

Format

EXE$KP_ALLOCATE_KPB kpb ,stack_size ,flags ,param_size

Context

EXE$KP_ALLOCATE_KPB conforms to the OpenVMS Alpha calling standard.

Because EXE$KP_ALLOCATE_KPB raises IPL to IPL$_SYNCH and obtains the MMG spin lock, its caller cannot be executing above IPL$_SYNCH or hold any higher ranked spin locks. EXE$KP_ALLOCATE_KPB returns control to its caller at its caller's IPL. The caller retains any spin locks it held at the time of the call.

Arguments

kpb

VMS Usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of KPB.

stack_size

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Requested size (in bytes) of kernel process stack.

flags

VMS Usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by value

Flags indicating the type, size, and configuration of the KPB to be created. EXE$KP_ALLOCATE_KPB accepts only the following flags:

KPB$V_VEST	KPB must be a VEST KPB. (See Chapter 10 for a description of VEST KPBs.)

KPB$V_SPLOCK	Spinlock area must be present. (Note that EXE$KP_ALLOCATE_KPB automatically sets this bit when KPB$V_VEST is set.)
KPB$V_DEBUG	Debug area must be present.
KPB$V_DEALLOC_AT_ END	KP_END should call KP_DEALLOCATE.

param_size

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Size in bytes of KPB parameter area, if any.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_BADPARAM	An illegal value was specified in the **flags** argument.
SS$_INSFARG	Not all of the required arguments were specified.
SS$_INSFMEM	KPB cannot be allocated because of a failure in the nonpaged pool allocation routine.
SS$_INSFRPGS	Kernel process stack cannot be allocated because of there are not enough free pages in the system.

Description

EXE$KP_ALLOCATE_KPB creates the KPB and the kernel process stack needed by a kernel process. It performs the following tasks:

- Verifies the contents of the **flags** parameter. If the **flags** parameter is valid, EXE$KP_ALLOCATE_KPB uses it as the basis for the mask it writes to KPB$IS_FLAGS. It automatically sets KPB$V_SCHED for all KPBs and, for VEST KPBs, also sets KPB$V_SPLOCK. Finally, it sets KPB$V_PARAM if a non-zero **param_size** argument is specified.
- Computes the size of the KPB to be allocated. For both VEST and non-VEST KPBs, the KPB includes the base KPB and scheduling area. VEST KPBs also, by default, include the spinlock area, which is optional for non-VEST KPBs. For VEST and non-VEST KPBs alike, the debug and parameter areas are optional. The presence of KP$V_DEBUG in the **flags** argument causes EXE$KP_ALLOCATE_KPB to include the KPB debug area; the presence of a non-zero **param_size** argument causes it to include the KPB parameter area (rounded up to an integral number of quadwords).

- Allocates a KPB of the appropriate size. If the KPB cannot be allocated, it returns SS$_INSFMEM status to its caller.
- Initializes the following KPB fields:

KPB$IB_TYPE	DYN$C_MISC
KPB$IB_SUBTYPE	DYN$C_KPB
KPB$IS_FLAGS	Computed flags value
KPB$PS_SCH_PTR	Address of KPB scheduling area
KPB$PS_SPL_PTR	Address of KPB spinlock area, if present
KPB$PS_DBG_PTR	Address of KPB debug area, if present
KPB$PS_PRM_PTR	Address of KPB parameter area, if present
KPB$IS_PRM_LENGTH	Length of the KPB parameter area, if specified, rounded up to an integral number of quadwords

- Computes the size of the kernel process stack by rounding the value of **stack_size** up to an integral number of CPU-specific pages, converting the result to bytes, and storing it in KPB$IS_STACK_SIZE.
- Allocates and initializes sufficient system PTEs for the stack, plus two no-access guard pages. If the sufficient PTEs are not available, EXE$KP_ALLOCATE_KPB deallocates the KPB and returns SS$_INSFRPGS status to its caller.
- Stores in KPB$PS_STACK_BASE the system virtual address of the start of the no-access guard page at the base of the kernel process stack. The kernel process stack grows negatively from this address.
- Inserts the address of the KPB in the location specified by the **kpb** argument.

The caller of EXE$KP_ALLOCATE_KPB is responsible for providing wait and retry operations in case of allocation failures.

EXE$KP_DEALLOCATE_KPB

Deallocates a KPB and its associated kernel process stack.

Module

KERNEL_PROCESS_MIN, KERNEL_PROCESS_MON

Macro

KP_DEALLOCATE_KPB

Format

EXE$KP_DEALLOCATE_KPB kpb

Context

EXE$KP_DEALLOCATE_KPB conforms to the OpenVMS Alpha calling standard.

EXE$KP_DEALLOCATE_KPB forks to perform KPB cleanup and call the routines that deallocate the KPB and the kernel process stack. As a result, drivers can call EXE$KP_DEALLOCATE_KPB from any IPL.

Arguments

kpb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of KPB.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_INSFARG	The **kpb** argument was not specified.

Description

EXE$KP_DEALLOCATE_KPB deallocates the KPB and the associated kernel process stack It performs the following tasks:

- Validates the structure indicated by the **kpb** argument. If the structure is not a KPB, or if it is currently valid, active, or in the process of deletion, EXE$KP_DEALLOCATE_KPB requests an INCONSTATE bugcheck.

- Indicates that KPB deletion is in progress by setting KPB$V_DELETING in KPB$IS_FLAGS.
- Sets up the KPB fork block (at KPB$PS_FQFL) so that the rest of KPB cleanup can transpire at IPL$_QUEUEAST. EXE$KP_DEALLOCATE_KPB issues a call to IOC$PRIMITIVE_FORK to queue the fork block on the IPL$_QUEUEAST fork queue. When IOC$PRIMITIVE_FORK returns control, EXE$KP_DEALLOCATE_KPB returns SS$_NORMAL status to its caller.
- When execution resumes at IPL$_QUEUEAST, the EXE$KP_DEALLOCATE_KPB fork routine deallocates the stack and returns the KPB to nonpaged pool.

EXE$KP_END

Terminates the execution of a kernel process.

Module

KERNEL_PROCESS_MAGIC

Macro

KP_END

Format

EXE$KP_END kpb

Context

EXE$KP_END conforms to the OpenVMS Alpha calling standard.

The caller of EXE$KP_END must be executing at IPL$_RESCHED or above.

Arguments

kpb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of KPB.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_INSFARG	The **kpb** argument was not specified.

Description

EXE$KP_END performs the following tasks to terminate the execution of a kernel process:

- If the **kpb** argument is not supplied, returns SS$_INSFARG status to its caller.
- Validates the structure indicated by the **kpb** argument. If the structure is not a KPB, or if it is currently invalid or inactive, EXE$KP_END requests an INCONSTATE bugcheck.

- Restores the SP of the initiator of the kernel process thread from KPB$PS_SAVED_SP and poisons that field.
- Restores the preserved registers (as indicated by KPB$IS_REG_MASK) and SP of the initiator of the kernel process thread.
- Marks the kernel process as inactive and invalid by clearing KPB$V_ACTIVE and KPB$V_VALID in KPB$IS_FLAGS.
- If KPB$V_DEALLOC_AT_END in KPB$IS_FLAGS is set (as it is in VEST KPBs), call EXE$KP_DEALLOCATE_KPB to deallocate the KPB and its associated kernel process stack.
- Returns successfully to the initiator of the kernel process thread (that is, the caller of EXE$START_KP or EXE$RESTART_KP).

EXE$KP_FORK

Stalls a kernel process in such a manner that it can be resumed by the OpenVMS fork dispatcher.

Module

KERNEL_PROCESS_MIN, KERNEL_PROCESS_MON

Macro

KP_STALL_FORK, KP_STALL_IOFORK

Format

EXE$KP_FORK kpb [,fkb]

Context

EXE$KP_FORK conforms to the OpenVMS Alpha calling standard. It can only be called by a kernel process.

Arguments

kpb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of the caller's KPB (which must be a VEST KPB). KPB$PS_UCB must contain the address of a UCB and KPB$PS_IRP must contain the address of an IRP.

fkb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of a fork block, usually in the UCB. If this argument is omitted, EXE$KP_FORK uses the fork block within the KPB (KPB$PS_FQFL).

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_BADPARAM	The **kpb** argument does not specify a VEST KPB.
SS$_INSFARG	Not all of the required arguments were specified.

Description

EXE$KP_FORK performs the following tasks in stalling the kernel process:

1. Saves the **kpb** argument in KPB$PS_FKBLK. If this argument is not specified to EXEKP_FORK, EXEKP_FORK writes the address of KPB$PS_FQFL into KPB$PS_FKBLK.
2. Inserts the procedure descriptor of subroutine STALL_FORK in KPB$PS_SCH_STALL_RTN, thus making it the kernel process scheduling stall routine.
3. Clears KPB$PS_SCH_RESTART, thus indicating that there is no kernel process scheduling restart routine.
4. Calls EXE$KP_STALL_GENERAL, passing to it the address of the KPB.

Having stalled the kernel process, the STALL_FORK kernel process scheduling stall routine returns control to EXE$KP_STALL_GENERAL, which returns to the initiator of the kernel process thread (that is, the caller of EXE$KP_START or EXE$KP_RESTART). When the fork dispatcher ultimately resumes the suspended routine, STALL_FORK calls EXE$KP_RESTART which, in turn, passes control back to EXE$KP_FORK. The kernel process forking stall routine then returns to the kernel process that called it.

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.

Module

KERNEL_PROCESS_MIN, KERNEL_PROCESS_MON

Macro

KP_STALL_FORK_WAIT

Format

EXE$KP_FORK_WAIT kpb [,fkb]

Context

EXE$KP_FORK_WAIT conforms to the OpenVMS Alpha calling standard and can only be called by a kernel process.

The caller of EXE$KP_FORK_WAIT must be executing at or above IPL$_SYNCH.

Arguments

kpb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of the caller's KPB.

fkb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of a fork block. If this argument is omitted, EXE$KP_FORK_WAIT uses the fork block within the KPB (KPB$PS_FKBLK).

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_INSFARG	Not all of the required arguments were specified.

Description

EXE$KP_FORK_WAIT performs the following tasks in stalling a kernel process:

1. Saves the **fkb** argument, if specified, in KPB$PS_FKBLK. If the argument is not specified, EXE$KP_FORK_WAIT moves the address of KPB$PS_FQFL into KPB$PS_FKBLK.
2. Inserts the procedure descriptor of subroutine STALL_FORK_WAIT in KPB$PS_SCH_STALL_RTN, thus making it the kernel process scheduling stall routine.
3. Clears KPB$PS_SCH_RESTART, thus indicating that there is no kernel process scheduling restart routine.
4. Calls EXE$KP_STALL_GENERAL, passing to it the address of the KPB.

Note that, having stalled the kernel process, the STALL_FORK_WAIT kernel process scheduling stall routine returns control to EXE$KP_STALL_GENERAL, which returns to the initiator of the kernel process thread (that is, the caller of EXE$KP_START or EXE$KP_RESTART). When the fork block is ultimately removed from the fork-and-wait-queue, STALL_FORK_WAIT calls EXE$KP_RESTART which, in turn, passes control back to EXE$KP_FORK_WAIT. EXE$KP_FORK_WAIT then returns to kernel process that called it.

EXE$KP_RESTART

Resumes the execution of a kernel process.

Module

KERNEL_PROCESS_MAGIC

Macro

KP_RESTART

Format

EXE$KP_RESTART kpb [,thread_status]

Context

EXE$KP_RESTART conforms to the OpenVMS Alpha calling standard.

The caller of EXE$KP_RESTART, usually a kernel process scheduling stall routine, must be executing at IPL$_RESCHED or above.

Arguments

kpb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of KPB.

thread_status

VMS Usage:	longword (unsigned)
type:	read only
access:	by value
mechanism:	

Status value to be returned to the kernel process that is to be resumed. This is the status returned by the call to EXE$KP_STALL_GENERAL. If the **thread_status** argument is not present, EXE$KP_RESTART returns SS$_NORMAL status to the kernel process.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_INSFARG	The **kpb** argument was not specified.

Description

EXE$KP_RESTART performs the following tasks to restart a kernel process:

1. Validates the structure indicated by the **kpb** argument. If the structure is not a KPB, or if it is currently invalid, EXE$KP_START requests an INCONSTATE bugcheck.
2. Preserves the current context by saving the current stack pointer (SP) and the registers indicated by KPB$IS_REG_MASK on the stack (which it quadword-aligns after obtaining the current SP). It saves the new value of the SP in KPB$PS_SAVED_SP.
3. Restores the SP of the stalled kernel process from KPB$PS_STACK_SP.
4. Restores the preserved registers (as indicated by KPB$IS_REG_MASK) from the top of the kernel process stack, plus the original SP of the kernel process stack.
5. Makes the KPB active by setting the corresponding bit in KPB$IS_FLAGS.
6. Calls the kernel process scheduling restart routine, if one is specified, passing it the KPB address, the return status value, and the procedure value of the kernel process spinlock restart routine.
7. Resumes the stalled kernel process.

EXE$KP_STALL_GENERAL

Stalls the execution of a kernel process.

Module

KERNEL_PROCESS_MAGIC

Macro

KP_STALL_GENERAL
KP_STALL_FORK
KP_STALL_FORK_WAIT
KP_STALL_IOFORK
KP_STALL_REQCHAN
KP_STALL_WFIKPCH
KP_STALL_WFIRLCH

Format

EXE$KP_STALL_GENERAL kpb

Context

EXE$KP_STALL_GENERAL conforms to the OpenVMS Alpha calling standard and can only be called by a kernel process.

Arguments

kpb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of the caller's KPB.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_INSFARG	Not all of the required arguments were specified.
Other values	As supplied to EXE$KP_RESTART

Description

EXE$KP_STALL_GENERAL suspends execution of the current kernel process. It performs the following tasks:

- Validates the structure indicated by the **kpb** argument. If the structure is not a KPB, or if it is currently valid, active, or in the process of deletion, EXE$KP_STALL_GENERAL requests an INCONSTATE bugcheck.
- Preserves the current context by saving the current kernel process stack pointer (SP) and the registers indicated by KPB$IS_REG_MASK on the stack (which it quadword-aligns after obtaining the current SP). It saves the new value of the kernel process SP in KPB$PS_STACK_SP.
- Restores the SP of the initiator of the kernel process thread from KPB$PS_SAVED_SP and poisons that field.
- Restores the preserved registers (as indicated by KPB$IS_REG_MASK) from the top of the initiator's stack, plus the original SP of the initiator of the kernel process thread.
- Marks the kernel process as inactive by clearing KPB$V_ACTIVE in KPB$IS_FLAGS.
- Calls the kernel process scheduling stall routine indicated by the procedure value in KPB$PS_SCH_STALL_RTN, passing it the KPB address and the procedure value of the spin lock stall handling routine (from KPB$PS_SPL_STALL_ROUTINE), or zero if the KPB spin lock area is not present. If there is no kernel process scheduling stall routine, EXE$KP_STALL_GENERAL requests an INCONSTATE bugcheck.

OpenVMS provides the following jacket routines for EXE$KP_STALL_GENERAL that supply scheduling stall routines for basic device driver functions:

Table 9–2 Kernel Process Stall Jacket Routines and Scheduling Stall Routines

Stall Jacket Routine	Scheduling Stall Routine[1]	Action of Stall Routine
EXE$KP_FORK	STALL_FORK	Calls EXE$PRIMITIVE_FORK on behalf of a kernel process. When it regains control from the OpenVMS fork dispatcher, this stall routine resumes the kernel process by calling EXE$KP_RESTART.
EXE$KP_FORK_WAIT	STALL_FORK_WAIT	Calls EXE$PRIMITIVE_FORK_WAIT on behalf of a kernel process. When it regains control from the OpenVMS software timer interrupt service routine (which resumes the entries on the fork-and-wait queue), this stall routine resumes the kernel process by calling EXE$KP_RESTART.

---

[1] These scheduling stall routines are not globally accessible.

(continued on next page)

Table 9–2 (Cont.) Kernel Process Stall Jacket Routines and Scheduling Stall Routines

Stall Jacket Routine	Scheduling Stall Routine[1]	Action of Stall Routine
EXE$KP_IOFORK	STALL_FORK	Calls EXE$PRIMITIVE_FORK (with timeouts disabled from the device unit associated with the KPB [UCB$PS_UCB]) on behalf of a kernel process. When it regains control from the OpenVMS fork dispatcher, this stall routine resumes the kernel process by calling EXE$KP_RESTART.
IOC$KP_REQCHAN	STALL_REQCHAN	Calls EXE$PRIMITIVE_REQCHAN on behalf of a kernel process. When it regains control after the channel has been granted, this stall routine resumes the kernel process by calling EXE$KP_RESTART.
IOC$KP_WFIKPCH	STALL_WFIXXCH	Issues the WFIKPCH macro on behalf of a kernel process. When it regains control due to a timeout or from interrupt servicing, this stall routine resumes the kernel process by calling EXE$KP_RESTART, returning to it SS$_NORMAL or SS$_TIMEOUT status.
IOC$KP_WFIRLCH	STALL_WFIXXCH	Issues the WFIRLCH macro on behalf of a kernel process. When it regains control due to a timeout or from interrupt servicing, it resumes the kernel process by calling EXE$KP_RESTART, returning to it SS$_NORMAL or SS$_TIMEOUT status.

[1] These scheduling stall routines are not globally accessible.

When the kernel process scheduling stall routine returns control, EXE$KP_STALL_GENERAL returns SS$_NORMAL status to the initiator of the kernel process thread (that is, the caller if EXE$KP_START or EXE$KP_RESTART).

EXE$KP_START

Starts the execution of a kernel process.

Module

KERNEL_PROCESS_MAGIC

Macro

KP_START
DDTAB (**start**=EXE$KP_STARTIO)

Format

EXE$KP_START kpb ,routine [,reg-mask]

Context

EXE$KP_START conforms to the OpenVMS Alpha calling standard. Its caller must be executing at IPL$_RESCHED or above.

Neither the initiator of the kernel process thread 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.

Arguments

kpb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of KPB.

routine

VMS Usage:	procedure_value
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Procedure value of the routine to be started as the top-level routine in the kernel process.

reg-mask

VMS Usage:	mask_quadword
type:	quadword (unsigned)
access:	read only
mechanism:	by value

Optional register save mask, indicating which registers must be preserved across kernel process context switches. Registers R0, R1, R16 through R25, R28, R30, and R31 (KPREG$K_ERR_REG_MASK) are never preserved across context switches; a **reg-mask** that indicates any of these registers is illegal. Registers R12 through R15, R26, R27, and R29 (KPREG$K_MIN_REG_MASK) are always saved and need not be specified.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_BADPARAM	An illegal **reg-mask** was specified.
SS$_INSFARG	Not all of the required arguments were specified.

Description

EXE$KP_START performs the following tasks to create a kernel process and start its execution:

1. Validates the structure indicated by the **kpb** argument. If the structure is not a KPB, or if it is currently valid, active, or in the process of deletion, EXE$KP_START requests an INCONSTATE bugcheck.
2. Constructs the register save mask from the value specified in **reg-mask**, if present, and the minimal register save mask. EXE$KP_START writes a value into this field that reflects the register save mask specified by its caller, plus a set of registers that are always preserved across such context switches (KPB$K_MIN_REG_MASK), including R12 through R15, R27, and R29.

   If an illegal **reg-mask** is specified, EXE$KP_START returns SS$_BADPARAM status to its caller. Otherwise, EXE$KP_START saves the register save mask in KPB$IS_REG_MASK.
3. Preserves the current context by saving the current stack pointer (SP) and the registers indicated by KPB$IS_REG_MASK on the stack (which it quadword-aligns after obtaining the current SP). It saves the new value of the SP in KPB$PS_SAVED_SP.
4. Establishes kernel process context by loading the base of the kernel process stack (KPB$PS_STACK_BASE) into the SP and KPB$PS_STACK_SP.
5. Makes the KPB active and valid by setting the corresponding bits in KPB$IS_FLAGS.
6. Initializes the bottom of the kernel process stack to enable implicit kernel process termination (by means of a call to EXE$KP_END) if the top-level kernel process routine returns to EXE$KP_START.
7. Calls the top-level kernel process routine, as indicated by the **routine** argument, passing to it the address of the KPB.

If the initiator of the kernel process thread and the kernel process must exchange additional parameters, they can do so only by using the KPB parameter area. The KPB parameter area is optionally created in the KPB by EXE$KP_ALLOCATE_KPB.

8. When it regains control as the result of the kernel process invoking the KP_REQCOM macro, calls EXE$KP_END.

EXE$KP_STARTIO

Sets up and starts a kernel process to be used by a device driver.

Module

KERNEL_PROCESS_MIN, KERNEL_PROCESS_MON

Macro

DDTAB (**start**=EXE$KP_STARTIO)

Format

JSB G^EXE$KP_STARTIO

Context

The caller of EXE$KP_STARTIO (usually IOC$INITIATE) must be executing at fork IPL and hold the fork lock indicated by UCB$B_FLCK. EXE$KP_STARTIO returns to its caller in fork context with no explicit output values.

Input

Location	**Contents**
R0	Address of DDT
R3	Address of IRP
R5	Address of UCB
UCB$L_BCNT	Number of bytes to be transferred
UCB$L_BOFF	Byte offset into first page of direct-I/O transfer; for buffered-I/O transfers, number of bytes to be charged to the process allocating the buffer
UCB$L_SVAPTE	For a direct-I/O transfer, virtual address of first page-table entry (PTE) of I/O-transfer buffer; for buffered-I/O transfer, addess of buffer is system address space
DDT$PS_KP_STARTIO	Procedure value of the driver's start-I/O routine, which serves as the top-level routine within the kernel process thread.
DDT$IS_STACK_BCNT	Size in bytes of the kernel process stack
DDT$IS_REG_MASK	Kernel process register save mask

Description

EXE$KP_STARTIO uses information stored in the DDT to set up and start a kernel process that can be used by a device driver. It performs the following tasks:

1. Establishes the size of the kernel process stack as the minimum of DDT$IS_STACK_BCNT and KPB$K_MIN_IO_STACK (currently 8KB).

2. Issues a standard call to EXE$KP_ALLOCATE_KPB to create the KPB and allocate the kernel process stack, passing to it the following:
   - Zero as the size of the KPB parameter area
   - KPB flags, indicating a VEST KPB with scheduling and spinlock areas, that is deallocated when the kernel process is terminated.
   - the kernel process stack size
   - IRP$PS_KPB as the target location of the KPB address

   If there were not enough free pages in the system for the kernel process stack, and the I/O request described by the IRP has not since been cancelled, EXE$KP_STARTIO issues a fork-and-wait request. When EXE$TIMEOUT resumes EXE$KP_STARTIO, it retries the call to EXE$KP_ALLOCATE_KPB.

   If the attempt to allocated nonpaged pool for the KPB failed, EXE$KP_STARTIO requests an INCONSTATE bugcheck.
3. Inserts the address of the IRP in KPB$PS_IRP and the address of the UCB in KPB$PS_UCB
4. Establishes the kernel process register save mask as the logical-OR of the registers specified in DDT$IS_REG_MASK and those indicated by KPREG$K_MIN_IO_REG_MASK (R2 through R5; the VAX AP, FP, SP, and PC [registers R12 through R15]; and R26, R27, and R29), minus those indicated by KPREG$K_ERR_REG_MASK (R0 and R1; R16 through R25; R28; R30; and R31).
5. Issues a standard call to EXE$KP_START, passing it the register save mask, the procedure value of a kernel process start-I/O routine (DDT$PS_KP_STARTIO), and the address of the KPB.
6. Issues an RSB instruction to its caller (usually IOC$INITIATE, or EXE$TIMEOUT if EXE$KP_STARTIO was resumed by fork-and-wait mechanism)

EXE$TIMEDWAIT_COMPLETE

Dermines whether the time interval of a timed wait has concluded.

Module

[SYSLOA]TIMEDWAIT

Macro

TIMEDWAIT

Format

EXE$TIMEDWAIT_COMPLETE end-value

Context

EXE$TIMEDWAIT_COMPLETE conforms to the OpenVMS Alpha calling standard.

Arguments

end-value

VMS Usage:	aligned quadword
type:	quadword (unsigned)
access:	modify
mechanism:	by reference

End time calculated by a previous call to EXE$TIMEDWAIT_SETUP or EXE$TIMEDWAIT_SETUP_10US.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_CONTINUE	The timed wait has not yet completed. The time interval for the timed wait may or may not have expired. This is a success status.
SS$_INSFARG	Not all of the required arguments were specified.
SS$_TIMEOUT	The time interval for a timed wait has expired and the timed wait is complete.

Description

EXE$TIMEDWAIT_COMPLETE compares the specified **end-value** (as computed by a prior call to EXE$TIMEDWAIT_SETUP or EXE$TIMEDWAIT_SETUP_10US) with an internal current-value. There are three results of this comparison:

- If the **end-value** is greater than or equal to the current-value value, the timed wait has not yet completed, and EXE$TIMEDWAIT_COMPLETE returns SS$_CONTINUE status.
- If the **end-value** is less than the current-value, EXE$TIMEDWAIT_COMPLETE sets the **end-value** to –1 and returns SS$_CONTINUE status.

  When EXE$TIMEDWAIT_COMPLETE returns SS$_CONTINUE status to the TIMEDWAIT macro, the macro reexecutes a specified series of instructions that tests for a particular exit condition. Having set the **end-value** to –1 prior to returning SS$_CONTINUE status, EXE$TIMEDWAIT_COMPLETE allows for the possibility that the exit condition was actually met during the timed wait time interval, but after the embedded instruction series could detect it. This could be the case, for instance, if an interrupt occurred and was serviced after the instruction sequence was executed but before the call to EXE$TIMEDWAIT_COMPLETE was made. As a result of this behavior, all timed wait instruction loops execute one additional time after the timed wait time interval has concluded.
- If the **end-value** is equal to –1, the timed wait has completed and EXE$TIMEDWAIT_COMPLETE returns SS$_TIMEOUT status.

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.

Module

[SYSLOA]TIMEDWAIT

Macro

TIMEDWAIT

Format

EXE$TIMEDWAIT_SETUP delta-time ,end-value

EXE$TIMEDWAIT_SETUP_10US delta-time ,end-value

Context

EXE$TIMEDWAIT_SETUP and EXE$TIMEDWAIT_SETUP_10US conform to the OpenVMS Alpha calling standard.

Arguments

delta-time

VMS Usage:	aligned quadword
type:	quadword (unsigned)
access:	read only
mechanism:	by reference

Delay time specified in nanoseconds (for EXE$TIMEDWAIT_SETUP) or 10-microsecond units (for EXE$TIMEDWAIT_SETUP_10US))

end-value

VMS Usage:	aligned quadword
type:	quadword (unsigned)
access:	write only
mechanism:	by reference

End time token to be supplied as input to EXE$TIMEDWAIT_COMPLETE.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_INSFARG	Not all of the required arguments were specified.

Description

EXE$TIMEDWAIT_SETUP and EXE$TIMEDWAIT_SETUP_10US compute the **end-value** that is supplied as an input argument to a subsequent call to EXE$TIMEDWAIT_COMPLETE. EXE$TIMEDWAIT_COMPLETE uses the **end-value** to determine whether the timed wait time interval has concluded.

EXE$TIMEDWAIT_SETUP and EXE$TIMEDWAIT_SETUP_10US generate a system-specific **end-value** from the sum of the specified **delta-time** and the current time, converted to a value that can be directly compared to an internal current-value. EXE$TIMEDWAIT_SETUP_10US performs the additional step of converting the input **delta-time** to a number of nanoseconds.

EXE_STD$ABORTIO

Completes the servicing of an I/O request without returning status to the I/O status block specified in the request.

Module

SYSQIOREQ

Format

status = EXE_STD$ABORTIO (irp, pcb, ucb, qio_sts)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
qio_sts	integer	input	value	required

irp
I/O request packet. EXE_STD$ABORTIO copies the **qio_sts** parameter to IRP$L_IOST1 and clears IRP$PS_FDT_CONTEXT. The caller of EXE_STD$ABORTIO should not access the IRP after the routine returns SS$_FDT_COMPL status.

pcb
PCB of current process

ucb
Unit control block

qio_sts
Final status to be returned by the $QIO system service to its caller. EXE_STD$ABORTIO places this status in FDT_CONTEXT$L_QIO_STATUS. If you intend to access the FDT context structure after EXE_STD$ABORTIO returns, you must obtain its address from IRP$PS_FDT_CONTEXT and store it before making the call.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

Contents of **qio_sts**
argument

Context

EXE_STD$ABORTIO executes at its caller's IPL and raises to fork IPL, acquiring the associated fork lock in a multiprocessing environment. As a result, its caller cannot be executing above fork IPL. A driver usually transfers control to EXE_STD$ABORTIO at IPL$_ASTDEL.

EXE_STD$ABORTIO returns to its caller at the caller's IPL.

Description

The FDT completion routine EXE_STD$ABORTIO terminates the servicing of an I/O request without returning status to the I/O status block specified in the original call to the $QIO system service.

EXE_STD$ABORTIO performs the following actions:

1. Examines the **qio_sts** argument. If the argument contains SS$_FDT_COMPL, EXE_STD$ABORTIO returns to its caller. This check prevents an I/O request from being aborted more than once.
2. Places the status to be returned to the caller of the $QIO system service in IRP$L_IOST1 and in the FDT_CONTEXT structure.
3. Clears the pointer to the FDT_CONTEXT structure in IRP$PS_FDT_CONTEXT.
4. Requests the fork lock, raising IPL to fork IPL, to perform the following tasks:
    a. Clear IRP$L_IOSB so that no status is returned by I/O postprocessing
    b. Clear ACB$V_QUOTA in IRP$B_RMOD to prevent the delivery of any AST to the process specified in the I/O request
    c. Update the count of available AST entries at PCB$L_ASTCNT, if necessary
    d. Insert the IRP in the local processor's I/O postprocessing queue. If the queue is empty, request a software interrupt from the local processor at IPL$_IOPOST.
5. Releases the fork lock, restoring the caller's IPL. The pending IPL$_IOPOST interrupt causes I/O postprocessing to occur before the remaining instructions in EXE_STD$ABORTIO are executed.

When all I/O postprocessing has been completed, EXE_STD$ABORTIO regains control and returns SS$_FDT_COMPL status to its caller.

Any ASTs specified when the I/O request was issued will not be delivered, and any event flags requested will not be set.

Macro

CALL_ABORTIO [do_ret=YES]

where:

> **do_ret** indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

In an Alpha driver, the CALL_ABORTIO macro initializes the **irp**, **pcb**, **ucb**, and **qio_sts** parameters from the contents of R3, R4, R5, and R0, respectively, and calls EXE_STD$ABORTIO. When EXE_STD$ABORTIO returns control to the code generated by a default invocation of $ABORTIO, a RET instruction returns control to the caller of $ABORTIO's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The order in which formal parameters are passed to EXE_STD$ABORTIO differs from the order in which they are provided in registers to the VAX routine EXE$ABORTIO.
- The contents of R0 are destroyed across the call to EXE_STD$ABORTIO. This is especially important if you use the $ABORTIO macro on OpenVMS Alpha systems and expect R0 to retain its value afterwards.
- Unlike EXE$ABORTIO, EXE_STD$ABORTIO does not lower IPL to 0 before exiting. EXE_STD$ABORTIO returns to its caller at the caller's IPL.
- EXE$ABORTIO returns control to the system service dispatcher, passing it the final $QIO system service status in R0. EXE_STD$ABORTIO returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$ALLOCBUF, EXE_STD$ALLOCIRP

Allocates a buffer from nonpaged pool for a buffered-I/O operation.

Module

MEMORYALC

Format

status = EXE_STD$ALLOCBUF (reqsize, blocksize, blockptr)

status = EXE_STD$ALLOCIRP (blocksize, blockptr)

Arguments

Argument	Type	Access	Mechanism	Status
reqsize	integer	input	value	required
alosize_p	pointer	output	value	required
bufptr_p	pointer	output	value	required

reqsize
Size of requested buffer in bytes (EXE_STD$ALLOCBUF only). This value should include the 12 bytes required to store header information.

alosize_p
Location in which EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP write the size of the requested buffer in bytes.

bufptr_p
Location in which EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP write the address of allocated buffer. The following fields are initialized in the buffer:

Field	Contents
IRP$W_SIZE (in allocated buffer)	Size of requested buffer in bytes (for EXE_STD$ALLOCBUF), IRP$C_LENGTH (for EXE_STD$ALLOCIRP).
IRP$B_TYPE (in allocated buffer)	DYN$C_BUFIO (for EXE_STD$ALLOCBUF), DYN$C_IRP (for EXE_STD$ALLOCIRP).

Return Values

SS$_NORMAL	Normal, successful completion.
SS$_INSFMEM	Insufficient memory to satisfy request.

Context

EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP set IPL to IPL$_ASTDEL. As a result they cannot be called by code executing above IPL$_ASTDEL. They return control to the caller at IPL$_ASTDEL.

Description

EXE_STD$ALLOCBUF attempts to allocate a buffer of the requested size from nonpaged pool; EXE_STD$ALLOCIRP attempts to allocate an IRP from nonpaged pool.

If sufficient memory is not available, EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP examine the PCB (CTL$GL_PCB) to determine whether the process has resource wait mode enabled. If PCB$V_SSRWAIT in PCB$L_STS is clear, these routines place the process in a resource wait state until memory is released.

The caller must check and adjust process quotas (JIB$L_BYTCNT or JIB$L_BYTLM, or both) by calling EXE$DEBIT_BYTCNT or EXE$DEBIT_BYTCNT_BYTLM.

Note

You can perform this task and allocate a buffer of the requested size by using the routines EXE$DEBIT_BYTCNT_ALO and EXE$DEBIT_BYTCNT_BYTLM_ALO. These routines invoke EXE_STD$ALLOCBUF.)

The normal buffered I/O postprocessing routine (IOC_STD$REQCOM), initiated by the REQCOM macro, readjusts quotas and also deallocates the buffer.

Note

The value returned in the **alosize_p** argument and placed at IRP$W_SIZE in the allocated buffer is the size of the allocated buffer. The actual size of the buffer is determined according to the algorithms used by EXE$ALONONPAGED and the size of the lookaside list packets. The nonpaged pool deallocation routine (EXE$DEANONPAGED), called in buffered I/O postprocessing, uses similar algorithms when returning memory to nonpaged pool.

Macro

CALL_ALLOCBUF
CALL_ALLOCIRP

In an Alpha driver, CALL_ALLOCBUF and CALL_ALLOCIRP simulate a JSB to EXE$ALLOCBUF and EXE$ALLOCIRP, respectively. CALL_ALLOCBUF calls EXE_STD$ALLOCBUF using the current contents of R1 as the **reqsize** argument. Both CALL_ALLOCBUF and CALL_ALLOCIRP return status in R0, the address of the allocated buffer in R2 and its size in R1. If a resource wait occurred, these macros return the address of the PCB in R4.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP replace EXE$ALLOCBUF and EXE$ALLOCIRP. The Alpha routines do not preserve the original contents of R4, or return the address of the PCB in R4 if a wait has occurred.

EXE_STD$ALTQUEPKT

Delivers an IRP to a driver's alternate start-I/O routine without regard for the status of the device.

Module

SYSQIOREQ

Format

EXE_STD$ALTQUEPKT (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet.

ucb
Unit control block.

EXE_STD$ALTQUEPKT reads the following UCB fields:

Field	Contents
UCB$B_FLCK	Fork lock index
UCB$L_DDT	Address of unit's DDT. EXE_STD$ALTQUEPKT reads DDB$PS_ALTSTART to obtain the procedure value of the driver's alternate start-I/O routine.
UCB$L_ALTIOWQ	Address of the alternate start-I/O wait queue listhead.

Context

A driver FDT routine typically calls EXE_STD$ALTQUEPKT at IPL$_ASTDEL. EXE_STD$ALTQUEPKT raises to fork IPL (acquiring the associated fork lock) before calling the driver's alternate start-I/O routine. When the alternate start-I/O routine returns control to it, EXE_STD$ALTQUEPKT returns control to its caller at the caller's IPL (having released its acquisition of the fork lock).

Description

EXE_STD$ALTQUEPKT calls the driver's alternate start-I/O routine. It does not test whether the unit is busy before making the call.

Macro

CALL_ALTQUEPKT

CALL_ALTQUEPKT calls EXE_STD$ALTQUEPKT, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$ALTQUEPKT replaces EXE$ALTQUEPKT.

EXE_STD$CARRIAGE

Interprets the carriage control specifier in IRP$B_CARCON and converts it to a generic prefix or suffix format.

Module

SYSQIOFDT

Format

EXE_STD$CARRIAGE (irp)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required

irp
I/O request packet.

Context

A driver FDT routine calls EXE_STD$CARRIAGE at IPL$_ASTDEL. EXE_STD$CARRIAGE returns control to the driver at that IPL.

Description

For Digital internal use only.

Macro

CALL_CARRIAGE

CALL_CARRIAGE calls EXE_STD$CARRIAGE, using the current contents of R3 as the **irp** arguments.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$CARRIAGE replaces EXE$CARRIAGE.

EXE_STD$CHK*xxx*ACCES

Checks logical (EXE_STD$CHKLOGACCES), physical (EXE_STD$CHKPHYACCES), read (EXE_STD$CHKRDACCES), write (EXE_STD$CHKWRTACCES), execute (EXE_STD$CHKEXEACCES), create (EXE_STD$CHKCREACCES), or delete (EXE_STD$CHKDELACCES) I/O function access, based on the specified protection information.

Module

EXSUBROUT

Format

status = EXE_STD$CHKCREACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKDELACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKEXEACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKLOGACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKPHYACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKRDACCES (arb, orb, pcb, ucb)

status = EXE_STD$CHKWRTACCES (arb, orb, pcb, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
arb	ARB	input	reference	required
orb	ORB	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required

arb
Agent rights block.

orb
Object rights block.

pcb
Process control block of accessor.

ucb
Unit control block of accessed object.

Return Values

SS$_NORMAL	Specified access allowed.
SS$_NOPRIV	Specified access denied.

Context

A driver FDT routine calls EXE_STD$CHKPHYACCES, EXE_STD$CHKLOGACCES, EXE_STD$CHKWRTACCES, EXE_STD$CHKRDACCES, EXE_STD$CHKCREACCES, EXE_STD$CHKEXEACCES, and EXE_STD$CHKDELACCES, at IPL$_ASTDEL. These routines return control to the driver at that IPL.

Description

For Digital internal use only.

Macro

```
CALL_CHKCREACCES [save_r1]
CALL_CHKDELACCES [save_r1]
CALL_CHKEXEACCES [save_r1]
CALL_CHKLOGACCES [save_r1]
CALL_CHKPHYACCES [save_r1]
CALL_CHKRDACCES [save_r1]
CALL_CHKWRTACCES [save_r1]
```

where:

> **save_r1** indicates that the macro must preserve the contents of R1 across the call to EXE_STD$CHKPHYACCES, EXE_STD$CHKLOGACCES, EXE_STD$CHKWRTACCES, EXE_STD$CHKEXEACCES, EXE_STD$CHKCREACCES, EXE_STD$CHKDELACCES or EXE_STD$CHKRDACCES. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

In an Alpha driver, the CALL_CHKCREACCES, CALL_CHKDELACCES, CALL_CHKEXEACCES, CALL_CHKLOGACCES, CALL_CHKPHYACCES, CALL_CHKWRTACCES, and CALL_CHKRDACCES, macros simulate the JSB to EXE$CHKCREACCES, EXE$CHKDELACCES, EXE$CHKEXEACCES, EXE$CHKPHYACCES, EXE$CHKLOGACCES, EXE$CHKWRTACCES, or EXE$CHKRDACCES in a VAX driver. Each macro calls the corresponding access-checking routine, using the current contents of R0, R1, R4, and R5 as the **arb**, **orb**, **pcb**, and **ucb** arguments. Unless you specify **save_r1**=**NO**, the macro preserves the quadword register R1 across the call. All macros return status in R0.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The OpenVMS Alpha I/O function access checking routines replace their OpenVMS VAX counterparts, but do not does not preserve R1 across a call.

EXE_STD$FINISHIO

Completes the servicing of an I/O request and returns status to the I/O status block specified in the original call to the $QIO system service.

Module

SYSQIOREQ

Format

status = EXE_STD$FINISHIO (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet. EXE_STD$FINISHIO clears IRP$PS_FDT_CONTEXT. The caller of EXE_STD$FINISHIO should not access the IRP after the routine returns SS$_FDT_COMPL status.

ucb
Unit control block. EXE_STD$FINISHIO increases UCB$L_OPCNT.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

EXE_STD$FINISHIO executes at its caller’s IPL and raises to fork IPL, acquiring the associated fork lock in a multiprocessing environment. As a result, its caller cannot be executing above fork IPL. A driver usually transfers control to EXE_STD$FINISHIO at IPL$_ASTDEL.

EXE_STD$FINISHIO returns to its caller at the caller’s IPL.

Description

The FDT completion routine EXE_STD$FINISHIO completes the servicing of an I/O request and returns status to the I/O status block specified in the original call to the $QIO system service. It performs the following actions:

1. Clears the pointer to the FDT context structure in IRP$PS_FDT_CONTEXT.
2. Requests the fork lock, raising IPL to fork IPL, to perform the following tasks:
    a. Increase the number of I/O operations completed on the current device in the operation count field of the UCB (UCB$L_OPCNT). This task is performed at fork IPL, holding the associated fork lock in a multiprocessing environment.
    b. Insert the IRP in the local processor's I/O postprocessing queue. If the queue is empty, request a software interrupt from the local processor at IPL$_IOPOST.
3. Releases the fork lock, restoring the caller's IPL. The pending IPL$_IOPOST interrupt causes I/O postprocessing to occur before the remaining instructions in EXE_STD$FINISHIO are executed.

When all I/O postprocessing has been completed, EXE_STD$FINISHIO regains control and returns SS$_FDT_COMPL status to its caller, passing SS$_NORMAL as the final $QIO completion status in the FDT_CONTEXT structure.

The image that requested the I/O operation receives SS$_NORMAL status, indicating that the I/O request has completed without device-independent error.

Macro

CALL_FINISHIO [do_ret=YES]
CALL_FINISHIOC [do_ret=YES]

where:

> **do_ret** indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

In an Alpha driver, the CALL_FINISHIO macro simulates the JMP to EXE$FINISHIO in the FDT routine of a VAX driver. The CALL_FINISHIOC macro simulates the JMP to EXE$FINISHIOC. The former macro moves the current contents of R0 and R1 into IRP$L_IOST1 and IRP$L_IOST2, respectively; the latter initializes IRP$L_IOST1 from R0 and clears IRP$L_IOST2. Both macros initialize the **irp** and **ucb** parameters from the contents of R3 and R5, respectively before calling EXE_STD$FINISHIO. When EXE_STD$FINISHIO returns control to the code generated by a default invocation of CALL_FINISHIO or CALL_FINISHIOC, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- A VAX driver supplies the first and second longwords of device-specific status (in R0 and R1) as input to EXE$FINISHIO. EXE$FINISHIO writes these longwords to IRP$L_IOST1 and IRP$L_IOST2, respectively, from which I/O postprocessing transfers their values to the I/O status block specified in the original $QIO call. These status longwords are not input parameters to EXE_STD$FINISHIO. Rather, an Alpha driver's FDT routine must fill in IRP$L_IOST1 and IRP$L_IOST2 before calling EXE_STD$FINISHIO.

  Because the OpenVMS VAX routines EXE$FINISHIO and EXE$FINISHIOC differ only in that the latter routine clears the second longword on I/O status, there is no Alpha equivalent of EXE$FINISHIOC. If the driver needs to clear the second I/O status longword, it simply does so before calling EXE_STD$FINISHIO.
- The address of the PCB, supplied as input to EXE$FINISHIO on OpenVMS VAX systems, is not provided as input to EXE_STD$FINISHIO.
- Unlike EXE$FINISHIO, EXE_STD$FINISHIO does not lower IPL to 0 before exiting. EXE_STD$FINISHIO returns to its caller at the caller's IPL.
- EXE$FINISHIO returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$FINISHIO returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status (SS$_NORMAL) in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE$ILLIOFUNC

Aborts I/O preprocessing for an I/O function not supported a driver.

Module

SYSQIOFDT

Format

status = EXE$ILLIOFUNC (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request

pcb
Process control block of the current process

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Context

FDT dispatching code in the $QIO system service calls EXE$ILLIOFUNC at IPL$_ASTDEL when processing an I/O function that is not supported by a driver. EXE$ILLIOFUNC returns to the system service dispatcher at IPL$_ASTDEL.

Description

Because any slot corresponding to an unsupported function in a driver's FDT action vector contains the procedure value of EXE$ILLIOFUNC, FDT dispatching code in the $QIO system service calls EXE$ILLIOFUNC to process any I/O request specifying an unsupported I/O function code.

EXE$ILLIOFUNC calls EXE_STD$ABORTIO to terminate the processing of the I/O request.

EXE_STD$INSERT_IRP

Inserts an I/O request packet (IRP) into the specified queue of IRPs according to the base priority of the process that issued the I/O request.

Module

SYSQIOREQ

Format

status = EXE_STD$INSERT_IRP (irp_lh, irp)

Arguments

Argument	Type	Access	Mechanism	Status
queue	listhead	input	reference	required
irp_lh	address	input	reference	required

irp_lh
I/O queue listhead for the device.

irp
I/O request packet. EXE_STD$INSERT_IRP reads the base address of the process requesting the I/O from IRP$B_PRI.

Return Values

status	Low bit set if at least one IRP is already in the queue, low bit clear if the IRP is the only entry.

Context

EXE_STD$INSERT_IRP must be called at fork IPL or higher. In an OpenVMS multiprocessing environment, the caller must also hold the associated fork lock. EXE_STD$INSERT_IRP does not alter IPL or acquire any spin locks. It returns to its caller.

Description

EXE_STD$INSERT_IRP determines the position of the specified IRP in the pending-I/O queue according to two factors:

- Priority of the IRP, which is derived from the requesting process's base priority as stored in the IRP$B_PRI
- Time that the entry is queued; for each priority, the queue is ordered on a first-in/first-out basis

EXE_STD$INSERT_IRP inserts the IRP into the queue at that position, adjusts the queue links, and returns a value to indicate the status of the queue.

Macro

CALL_INSERT_IRP

In an Alpha driver, the CALL_INSERT_IRP macro simulates a JSB to EXE$INSERT_IRP in a VAX driver. CALL_INSERT_IRP calls EXE_STD$INSERT_IRP, using the current contents of R2 and R3 as the **irp_lh** and **irp** arguments, respectively. It returns status in R0.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$INSERT_IRP replaces EXE$INSERTIRP (used by OpenVMS VAX drivers).

EXE_STD$INSIOQ, EXE_STD$INSIOQC

Insert an IRP in a device's pending-I/O queue and call the driver's start-I/O routine if the device is not busy.

Module

SYSQIOREQ

Format

EXE_STD$INSIOQ (irp, ucb)

EXE_STD$INSIOQC (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet.

ucb
Unit control block.

EXE_STD$INSIOQ and EXE_STD$INSIOQC read the following UCB fields:

Field	Contents
UCB$B_FLCK	Fork lock index
UCB$L_STS	UCB$V_BSY set if device is busy, clear if device is idle
UCB$L_IOQFL	Address of pending-I/O queue listhead
UCB$L_QLEN	Length of pending-I/O queue

EXE_STD$INSIOQ and EXE_STD$INSIOQC write the following UCB fields:

Field	Contents
UCB$L_STS	UCB$V_BSY set
UCB$W_QLEN	Increased

Context

EXE_STD$INSIOQ and EXE_STD$INSIOQC immediately raise to fork IPL and, in a multiprocessing environment, obtain the corresponding fork lock. As a result, their callers must not be executing at an IPL higher than fork IPL or hold a spin lock ranked higher than the fork lock.

EXE_STD$INSIOQ unconditionally releases ownership of the fork lock before returning control to the caller without possession of the fork lock. If a fork process must retain possession of the fork lock, it should call EXE_STD$INSIOQC instead.

Description

EXE_STD$INSIOQ and EXE_STD$INSIOQC insert an IRP in a device's pending-I/O queue and call the driver's start-I/O routine if the device is not busy.

EXE_STD$INSIOQ and EXE_STD$INSIOQC increase UCB$L_QLEN and proceed according to the status of the device (as indicated by UCB$V_BSY in UCB$L_STS) as follows:

- If the device is busy, call EXE_STD$INSERT_IRP to place the IRP on the device's pending-I/O queue.
- If the device is idle, call IOC_STD$INITIATE to begin device processing of the I/O request immediately. IOC_STD$INITIATE transfers control to the driver's start-I/O routine.

Macro

CALL_INSIOQ
CALL_INSIOQC

In an Alpha driver, the CALL_INSIOQ and CALL_INSIOQC macros simulate a JSB to EXE$INSIOQ and EXE$INSIOQ, respectively, in a VAX driver. $INSIOQ calls EXE_STD$INSIOQ, and $INSIOQC calls EXE_STD$INSIOQC, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

Notes for Converting VAX Drivers

None.

EXE_STD$IORSNWAIT

Places a process in a resource wait state if it has enabled resource waits.

Module

SYSQIOFDT

Format

status = EXE_STD$IORSNWAIT (irp, pcb, ucb, ccb, qio_sts, rsn)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required
qio_sts	input	integer	value	required
rsn	input	integer	value	required

irp
I/O request packet.

pcb
Process control block.

ucb
Unit control block.

ccb
Channel control block.

qio_sts
Final status to be returned by the $QIO system service to its caller if the caller has not enabled resource wait mode. EXE_STD$IORSNWAIT calls EXE_STD$ABORTIO to place this status in FDT_CONTEXT$L_QIO_STATUS. If you intend to access the FDT context structure after EXE_STD$IORSNWAIT returns, you must obtain its address from IRP$PS_FDT_CONTEXT and store it before making the call.

rsn
Number of the resource for which the request is waiting.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

Contents of **qio_sts** argument	Process has not enabled resource waits.
SS$_WAIT_CALLERS_MODE	Process has been placed in a resource wait state.

Context

EXE_STD$IORSNWAIT is called by, and returns to, a driver's FDT routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

```
CALL_IORSNWAIT [do_ret=YES]
```

where:

do_ret indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

In an Alpha driver, the CALL_IORSNWAIT macro calls EXE_STD$IORSNWAIT using the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **qio_sts**, and **rsn** arguments, respectively. When EXE_STD$IORSNWAIT returns control to the code generated by a default invocation of CALL_IORSNWAIT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$IORSNWAIT replaces EXE$IORSNWAIT. The order in which formal parameters are passed to EXE_STD$IORSNWAIT differs from the order in which they are provided in registers to the VAX routine EXE$IORSNWAIT.
- EXE$IORSNWAIT returns control to the system service dispatcher, passing it the final $QIO system service status in R0. EXE_STD$IORSNWAIT returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT context structure. The $QIO system service retrieves the status from this structure.

EXE_STD$LCLDSKVALID

Processes I/O functions that affect the online count and local valid status of a disk.

Module

SYSQIOFDT

Format

status = EXE_STD$LCLDSKVALID (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp

I/O request packet for the current I/O request. The I/O function for the current request is available in IRP$L_FUNC.

pcb

Process control block of the current process.

ucb

Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

EXE_STD$LCLDSKVALID reads the following UCB fields.

Field	Contents
UCB$B_FLCK	Fork lock index
UCB$L_STS	UCB$V_LCL_VALID set if the volume is valid; clear if the drive is unloaded or available
UCB$B_ONLCNT	Number of hosts that have set this disk on line

EXE_STD$LCLDSKVALID writes the following UCB fields:

Field	Contents
UCB$L_STS	UCB$V_LCL_VALID set if the requested function is IO$_PACKACK; cleared if the requested function is IO$_UNLOAD or IO$_AVAILABLE

Field	Contents
UCB$B_ONLCNT	Incremented if UCB$V_LCL_VALID is not set and the requested function is IO$_PACKACK; decremented if UCB$V_LCL_VALID is set and the requested function is IO$_UNLOAD or IO$_AVAILABLE

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

FDT dispatching code calls EXE_STD$LCLDSKVALID at IPL$_ASTDEL. EXE_STD$LCLDSKVALID immediately raises IPL to fork IPL, requesting the associated fork lock in a multiprocessing environment. When it regains control from EXE_STD$QIODRVPKT or EXE_STD$FINISHIO, EXE_STD$LCLDSKVALID lowers IPL to IPL$_ASTDEL and relinquishes the fork lock before returning to the system service dispatcher.

Description

A disk driver specifies the system-supplied upper-level FDT action routine EXE_STD$LCLDSKVALID in an FDT_ACT macro invocation to service a request for an IO$_PACKACK, IO$_AVAILABLE, or IO$_UNLOAD function for a local disk. The actions of EXE_STD$LCLDSKVALID depend on the I/O function indicated by R7 and the value of UCB$V_LCL_VALID in UCB$L_STS.

For an IO$_PACKACK function, EXE_STD$LCLDSKVALID proceeds as follows:

- If UCB$V_LCL_VALID is clear:
  - Sets UCB$V_LCL_VALID.
  - Increases UCB$B_ONLCNT.
  - If this is the first cluster pack acknowledgment on the disk (that is, if UCB$B_ONLCNT equals 1), invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$LCLDSKVALID regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.
- If UCB$V_LCL_VALID is set, EXE_STD$LCLDSKVALID requests that the FDT completion routine EXE_STD$FINISHIO complete the I/O request. EXE_STD$FINISHIO returns to EXE_STD$LCLDSKVALID with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.

For an IO$_UNLOAD or IO$_AVAILABLE function, EXE_STD$LCLDSKVALID proceeds as follows:

- If UCB$V_LCL_VALID is set:
    - Clears UCB$V_LCL_VALID
    - Decreases UCB$B_ONLCNT
    - If this is the last cluster unload or available request, invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$LCLDSKVALID regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.
- If UCB$V_LCL_VALID is clear, EXE_STD$LCLDSKVALID requests that the FDT completion routine EXE_STD$FINISHIO complete the I/O request. EXE_STD$FINISHIO returns to EXE_STD$LCLDSKVALID with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.

A driver must define the local disk UCB extension to use this routine.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The upper-level FDT routine EXE$LCLDSKVALID (used by OpenVMS VAX device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table (FDT) from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$LCLDSKVALID.

EXE_STD$MNTVERSIO

Initiates a mount verification I/O request to a device.

Module

MOUNTVER

Format

EXE_STD$MNTVERSIO (rout, irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
rout	procedure value	input	value	required
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

rout
Procedure value of action routine to postprocess the mount verification I/O request.

irp
I/O request packet.

ucb
Unit control block.

Context

EXE_STD$MNTVERSIO raises IPL to fork IPL, obtaining the corresponding fork lock in an OpenVMS multiprocessing system. It releases the fork lock and returns control to its caller at its caller's IPL.

Description

For Digital internal use only.

Macro

CALL_MNTVERSIO

In an Alpha driver, the CALL_MNTVERSIO macro calls EXE_STD$MNTVERSIO, using the current contents of R0, R3, and R5 as the **rout**, **irp**, and **ucb** arguments, respectively.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$MNTVERSIO replaces EXE$MNTVERSIO.

EXE_STD$MODIFY

Translates a logical read/write function into a physical read/write function, transfers $QIO system service parameters to the IRP, validates and prepares a user buffer, and aborts the request or proceeds with a direct-I/O, DMA read/write operation.

Module

SYSQIOFDT

Format

status = EXE_STD$MODIFY (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request.

EXE_STD$MODIFY reads the following IRP fields:

Field	Contents
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the buffer's virtual address.
IRP$L_QIO_P2	$QIO system service **p2** argument, containing the number of bytes in transfer. The maximum number of bytes that EXE_STD$MODIFY can transfer is 65,535 (128 pages minus one byte).
IRP$L_QIO_P4	$QIO system service **p4** argument, containing the carriage control byte.
IRP$L_FUNC	I/O function code.
IRP$B_RMOD	Access mode of the caller of the $QIO system service.

EXE_STD$MODIFY writes the following IRP fields:

Field	Contents
IRP$B_CARCON	Carriage control byte (from IRP$L_QIO_P4)
IRP$L_FUNC	Logical read/write function code converted to physical

Field	Contents
IRP$L_STS	IRP$V_FUNC set to indicate read function
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buffer** parameter does not allow read access.
SS$_BADPARAM	**size** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	The I/O request has been successfully queued.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$MODIFY as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$MODIFY to prepare a direct-I/O read/write request. A driver cannot specify EXE_STD$MODIFY for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their own upper-level FDT action routines to handle them.

EXE_STD$MODIFY performs the following functions:

- Sets IRP$V_FUNC in IRP$L_STS to indicate a read function
- Copies the **p4** argument of the $QIO request from IRP$L_QIO_P4 to IRP$B_CARCON

- Translates a logical read/write function to a physical read/write function and stores the new function code in IRP$L_FUNC.
- Examines the size of the transfer, as specified in the **p2** argument of the $QIO request (IRP$L_QIO_P2), and takes one of the following actions:
  - If the transfer byte count is zero, EXE_STD$MODIFY invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$MODIFY regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

    The driver start-I/O routine should check for zero-length buffers to avoid mapping to adapter node space. An attempted mapping can cause a system failure.
  - If the byte count is not zero, EXE_STD$MODIFY calls EXE_STD$MODIFYLOCK, passing 0 as the value of the **err_rout** argument.

EXE_STD$MODIFYLOCK disables an optimization in MMG_STD$IOLOCK and joins the code for EXE_STD$READLOCK. EXE_STD$MODIFYLOCK invokes the $READCHK macro, which calls EXE_STD$READCHK.

EXE_STD$READCHK performs the following actions:

- Moves the transfer byte count (**size** parameter) into IRP$L_BCNT.

  If the byte count is negative, it calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, EXE_STD$READCHK returns to EXE_STD$MODIFYLOCK with SS$_BADPARAM status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$MODIFYLOCK immediately returns to EXE_STD$MODIFY, passing these status values. EXE_STD$MODIFY, in turn, returns to the $QIO system service.
- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:
  - If the buffer allows write access returns SS$_NORMAL in R0 to EXE_STD$MODIFYLOCK.
  - If the buffer does not allow write access, EXE_STD$READCHK calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_ACCVIO. When it regains control, EXE_STD$READCHK returns to EXE_STD$MODIFYLOCK with SS$_ACCVIO status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$MODIFYLOCK immediately returns to EXE_STD$MODIFY, passing these status values. EXE_STD$MODIFY returns to the $QIO system service.

If EXE_STD$READCHK succeeds, EXE_STD$MODIFYLOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$MODIFYLOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$MODIFYLOCK. EXE_STD$MODIFYLOCK returns immediately to EXE_STD$MODIFY, passing to it this status value.

EXE_STD$MODIFY invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$MODIFY regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$MODIFYLOCK.

  For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$MODIFYLOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$MODIFYLOCK returns EXE_STD$MODIFY the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$MODIFY returns to the $QIO system service.

  For page fault status, EXE_STD$MODIFYLOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine EXE$MODIFY expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$MODIFY.

- EXE$MODIFY returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL, SS$_ACCVIO, or SS$_BADPARAM, or SS$_INSFWSL) in R0. EXE_STD$MODIFY returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$MODIFYLOCK

Validates and prepares a user buffer for a direct-I/O, DMA read/write operation.

Module

SYSQIOFDT

Format

status = EXE_STD$MODIFYLOCK (irp, pcb, ucb, ccb, buf, bufsiz, err_rout)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required
buf	address	input	reference	required
bufsiz	integer	input	value	required
err_rout	procedure value	input	value	required

irp
I/O request packet for the current I/O request.

EXE_STD$MODIFYLOCK reads IRP$B_RMOD to determine the access mode of the caller of the $QIO system service.

EXE_STD$MODIFYLOCK writes the following IRP fields:

Field	Contents
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

buf
Virtual address of buffer.

bufsiz
Number of bytes in transfer.

err_rout
Procedure value of error-handling callback routine, or 0 if the driver does not process errors.

A driver typically specifies an error-handling callback routine when the driver must lock multiple areas into memory for a single I/O request and regain control to unlock these areas, if the request is to be aborted. The routine performs those tasks required before the request is backed out of or aborted. Such operations could include calling MMG_STD$UNLOCK to release previous buffers participating in the I/O operation. The error-handling routine must preserve R0 and R1 and return back to EXE_STD$MODIFYLOCK.

Chapter 8 describes the error-handling callback routine interface.

Return Values

SS$_NORMAL	The buffer is read-accessible and has been locked in memory.
SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow read access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Nothing has occurred yet to prevent the I/O request from being successfully queued. This is the initial value of the status field in an FDT_CONTEXT structure.
SS$_INSFWSL	Insufficient working set limit.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

The system-supplied upper-level FDT action routine EXE_STD$MODIFY, or a driver-specific upper-level FDT action routine, calls EXE_STD$MODIFYLOCK at IPL$_ASTDEL.

Description

A driver FDT routine calls the system-supplied FDT support routine EXE_STD$MODIFYLOCK to check the read accessibility of an I/O buffer supplied in a $QIO request for a read/write function, and lock the buffer in memory in preparation for a DMA read/write operation.

A driver cannot specify EXE_STD$MODIFY for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their FDT routines to handle them.

EXE_STD$MODIFYLOCK disables an optimization in MMG_STD$IOLOCK and joins the code for EXE_STD$READLOCK. EXE_STD$MODIFYLOCK invokes the $READCHK macro, which calls EXE_STD$READCHK.

EXE_STD$READCHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, EXE_STD$READCHK returns SS$_BADPARAM status to EXE_STD$MODIFYLOCK.

- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:
  - If the buffer allows write access, EXE_STD$READCHK sets IRP$V_FUNC in IRP$L_STS and returns SS$_NORMAL in R0 to EXE_STD$MODIFYLOCK.
  - If the buffer does not allow write access, EXE_STD$READCHK returns SS$_ACCVIO status to EXE$_STD$MODIFYLOCK.

If error status (SS$_BADPARAM or SS$_ACCVIO) is returned, EXE_STD$MODIFYLOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$MODIFYLOCK. When the callback routine returns (or if no callback routine is specified), EXE_STD$MODIFYLOCK calls EXE_STD$ABORTIO, passing it the error status as **qio_sts**. EXE_STD$ABORTIO returns to EXE_STD$MODIFYLOCK with the error status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$MODIFYLOCK immediately returns to its caller, passing these status values.

If SS$_NORMAL status is returned, EXE_STD$MODIFYLOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$MODIFYLOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$MODIFYLOCK. EXE_STD$MODIFYLOCK returns immediately to its caller, passing to it this status value.
- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$MODIFYLOCK. EXE_STD$MODIFYLOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$MODIFYLOCK. When

the callback routine returns (or if no callback routine is specified), EXE_STD$MODIFYLOCK proceeds as follows:

- For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$MODIFYLOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$MODIFYLOCK returns to its caller the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.

  For page fault status, EXE_STD$MODIFYLOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

The caller of EXE_STD$MODIFYLOCK must examine the status in R0:

- If the status is SS$_NORMAL, the buffer is write accessible and has been successfully locked into memory and the starting virtual address of the page table entries that map the buffer is available in IRP$L_SVAPTE.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the I/O request to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS. Ordinarily a driver specifies an error-handling callback routine to process such errors.

  Note that a driver cannot access the IRP once it has received SS$_FDT_COMPL status. If you know you need access to information stored in the IRP to back out an I/O request that has been aborted, you must store that information elsewhere prior to calling EXE_STD$MODIFYLOCK.

Macro

CALL_MODIFYLOCK
CALL_MODIFYLOCK_ERR [interface_warning=YES]

where:

> **interface_warning**=**YES**, the default, specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the VAX version of the corresponding system routine. **interface_warning**=**NO** suppresses the warning.

In an Alpha driver, CALL_MODIFYLOCK simulates a JSB to EXE$MODIFYLOCK and CALL_MODIFYLOCK_ERR simulates a JSB to EXE$MODIFYLOCK_ERR. CALL_MODIFYLOCK calls EXE_STD$MODIFYLOCK, specifying 0 as the **err_rout** argument; CALL_MODIFYLOCK_ERR also calls EXE_STD$MODIFYLOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsiz** arguments, respectively.

When EXE_STD$MODIFYLOCK or EXE_STD$MODIFYLOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro moves the contents of IRP$L_SVAPTE into R1 and writes a 5 into R2 to indicate a modify operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro writes a 5 to R2 to indicate a modify operation and and returns to FDT dispatching code in the $QIO system service.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$MODIFYLOCK replaces EXE$MODIFYLOCK and EXE$MODIFYLOCK_ERR.

  R0, R7, and R8 are not provided as input to EXE_STD$MODIFYLOCK.
- The order in which formal parameters are passed to EXE_STD$MODIFYLOCK differs from the order in which they are provided in registers to the VAX routines EXE$MODIFYLOCK and EXE$MODIFYLOCK_ERR.
- EXE$MODIFYLOCK_ERR provides a mechanism by which a driver callback routine obtains control upon an error condition prior to the abortion of an I/O request. EXE_STD$MODIFYLOCK accepts the address of an error-handling callback routine in the **err_rout** argument. The error-handling routine is called after an I/O request encounters a buffer access or memory allocation failure and before the request is aborted.
- The design of FDT processing for OpenVMS Alpha device drivers guarantees that the caller of EXE_STD$MODIFYLOCK regains control whether the modify lock operation is successful or not. When a driver regains control from a call to EXE_STD$MODIFYLOCK, return status in R0 indicates that the buffer has been successfully locked (SS$_NORMAL) or that the operation failed and the request has been aborted (SS$_FDT_COMPL). The driver must check the return status and take appropriate action. Final $QIO completion status, indicating the reason the operation failed, is stored in the FDT_CONTEXT structure.

  Normally, a driver services a modify lock failure by supplying the address of an error-handling callback routine to EXE_STD$MODIFYLOCK.
- Driver code that executes after receiving failure status (SS$_FDT_COMPL) from EXE_STD$MODIFYLOCK cannot access information in the IRP. If the driver anticipates accessing IRP fields when EXE_STD$MODIFYLOCK returns, it must store these fields elsewhere before calling EXE_STD$MODIFYLOCK.
- Upon successful completion, EXE$MODIFYLOCK and EXE$MODIFYLOCK_ERR provide as output the system virtual address of the first process PTE that maps the buffer in R1 and in IRP$L_SVAPTE. Because EXE_STD$MODIFYLOCK does not provide R1 as output, a driver must obtain this information from IRP$L_SVAPTE. Similarly, the VAX routines set R2 to 1 to indicate a a read function. EXE_STD$MODIFYLOCK does not provide R2 as output; a driver can determine whether a function is write or read by examining IRP$V_FUNC in IRP$L_STS.

EXE_STD$MOUNT_VER

During I/O postprocessing, determines whether mount verification should be initiated on a given disk or tape device on behalf of the I/O request being completed.

Module

MOUNTVER

Format

status = EXE_STD$MOUNT_VER (iost1, iost2, irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
iost1	integer	input	value	required
iost2	integer	input	value	required
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

iost1
First longword of I/O status.

iost2
Second longword of I/O status.

irp
I/O request packet. This argument is 0 if there is no IRP to clean up.

ucb
Unit control block.

Return Values

status	Low bit set indicates that mount verification has not been initiated and that the caller should continue; low bit clear indicates that mount verification has been initiated and that the caller should return.

Context

EXE_STD$MOUNT_VER is typically called at or above IPL$_IOPOST.

Description

For Digital internal use only.

Macro

CALL_MOUNT_VER [save_r0r1]

where:

> **save_r0r1** indicates that the macro should preserve registers R0 and R1 across the call to EXE_STD$MOUNT_VER. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

In an Alpha driver, CALL_MOUNT_VER calls EXE_STD$MOUNT_VER, using the current contents of R0, R1, R3, and R5 as the **iost1**, **iost2**, **irp**, and **ucb** arguments, respectively. When EXE_STD$MOUNT_VER returns, code generated by this macro copies return status from R0 to R2. Unless you specify **save_r0r1**=**NO**, the macro preserves the quadword registers R0 and R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$MOUNT_VER replaces EXE$MOUNT_VER. Unlike EXE$MOUNT_VER, EXE_STD$MOUNT_VER does not preserve R0 and R1 across the call, or provide its return status in R2.

EXE_STD$ONEPARM

Copies a single $QIO parameter from IRP$L_QIO_P1 to IRP$L_MEDIA and delivers the IRP to a driver's start-I/O routine.

Module

SYSQIOFDT

Format

status = EXE_STD$ONEPARM (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request. EXE_STD$ONEPARM copies the first $QIO function-specific parameter (**p1**) from IRP$L_QIO_P1 to IRP$L_MEDIA.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$ONEPARM as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$ONEPARM to process an I/O function code that requires only one parameter. This parameter should need no checking: for instance, for read or write accessibility.

EXE_STD$ONEPARM copies the first $QIO function-dependent parameter (**p1**) from IRP$L_QIO_P1 to IRP$L_MEDIA and invokes the $QIODRVPKT macro to deliver the IRP to the driver. EXE_STD$ONEPARM regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine EXE$ONEPARM (used by OpenVMS VAX) obtains the value of the first function-dependent argument (**p1**) specified in the $QIO request from 00(AP). An OpenVMS Alpha FDT routine cannot obtain the argument as an offset from the AP; rather, it accesses the argument from a new IRP field, IRP$L_QIO_P1.

  In order to convert an OpenVMS VAX driver to an OpenVMS Alpha driver, the upper-level action routine EXE_STD$ONEPARM exists, to move the value of (**p1**) from IRP$L_QIO_P1 to IRP$L_MEDIA and invokes the $QIODRVPKT macro. If your driver does not access (**p1**) from IRP$L_MEDIA, but rather, uses the contents of IRP$L_QIO_P1, specifying EXE_STD$ONEPARM as an upper-level FDT action routine may defy all logic. (If your driver ignores the contents of IRP$L_MEDIA, it is immaterial whether you specify EXE_STD$ZEROPARM or EXE_STD$ONEPARM as the upper-level FDT action routine that delivers the IRP to the driver.) To avoid the unnecessary copy, you can write an upper-level FDT action routine that invokes the $QIODRVPKT macro.

- EXE$ONEPARM expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table (FDT) from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$ONEPARM.

- EXE$ONEPARM returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$ONEPARM returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$PRIMITIVE_FORK

Creates a simple fork process on the local processor.

Module

FORKCNTRL

Format

EXE_STD$PRIMITIVE_FORK (fr3, fr4, fkb)

Arguments

Argument	Type	Access	Mechanism	Status
fr3	int64	input	value	required
fr4	int64	input	value	required
fkb	FKB	input	reference	required

fr3
Value to pass to the fork routine in FKB$Q_FR3.

fr4
Value to pass to the fork routine in FKB$Q_FR4.

fkb
Fork block. At input, FKB$B_FLCK must contain the fork lock index and FKB$L_FPC must contain the procedure value of the fork routine.

Context

EXE_STD$PRIMITIVE_FORK acquires no spin locks and leaves IPL unchanged. EXE_STD$PRIMITIVE_FORK, unlike the OpenVMS VAX system routine EXE$FORK, returns to its caller and not to its caller's caller. It assumes that, prior to the call, its caller has placed the procedure value of the fork routine into FKB$L_FPC.

EXE_STD$PRIMITIVE_FORK provides fork context to the fork routine in FKB$Q_FR3 (contents of **fr3**) and FKB$Q_FR4 (contents **fr4**). All other registers are destroyed. The fork routine executes at the IPL indicated by the fork lock index stored in FKB$B_FLCK.

Description

EXE_STD$PRIMITIVE_FORK moves the contents of the **fr3** and **fr4** arguments into FKB$Q_FR3 and FKB$Q_FR4, respectively. It determines the fork IPL by using the value of FKB$B_FLCK as an index into the spin lock IPL vector (SMP$AL_IPLVEC). EXE_STD$PRIMITIVE_FORK inserts the fork block into the fork queue on the local processor (headed by CPU$Q_SWIQFL) corresponding to this IPL. If the queue is empty, EXE_STD$PRIMITIVE_FORK issues a SOFTINT macro, requesting a software interrupt from the local processor at that fork IPL.

A driver that calls EXE_STD$PRIMITIVE_FORK explicitly (that is, instead of invoking the IOFORK macro) must ensure that UCB$V_TIM in the UCB$L_STS field is clear before making the call.

Macro

FORK
IOFORK

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$PRIMITIVE_FORK is a call-based routine that performs the same operation as the JSB-based routine EXE$PRIMITIVE_FORK on OpenVMS Alpha systems. The OpenVMS VAX routines EXE$FORK and EXE$IOFORK are not provided on OpenVMS Alpha systems.

EXE_STD$PRIMITIVE_FORK_WAIT

Inserts a fork block on the fork-and-wait queue.

Module

FORKCNTRL

Format

EXE_STD$PRIMITIVE_FORK_WAIT (fr3, fr4, fkb)

Arguments

Argument	Type	Access	Mechanism	Status
fr3	int64	input	value	required
fr4	int64	input	value	required
fkb	FKB	input	reference	required

fr3
Value to pass to the fork routine in FKB$Q_FR3.

fr4
Value to pass to the fork routine in FKB$Q_FR4.

fkb
Fork block. At input, FKB$B_FLCK must contain the fork lock index and FKB$L_FPC must contain the procedure value of the fork routine.

Context

The caller of EXE_STD$PRIMITIVE_FORK_WAIT must be executing at or above IPL$_SYNCH. EXE_STD$PRIMITIVE_FORK_WAIT acquires the MEGA (SPL$C_MEGA) spin lock, raising IPL to IPL$_MEGA in the process, to access the fork-and-wait queue (EXE$AR_FORK_WAIT_QUEUE). It releases the spin lock, restoring the previous IPL, prior to returning to its caller.

EXE_STD$PRIMITIVE_FORK_WAIT, unlike the OpenVMS VAX system routine EXE$FORK_WAIT, returns to its caller and not to its caller's caller. It assumes that, prior to the call, its caller has placed the procedure value of the fork routine into FKB$L_FPC.

EXE_STD$PRIMITIVE_FORK_WAIT provides fork context to the fork routine in FKB$Q_FR3 (contents of **fr3**) and FKB$Q_FR4 (contents of **fr4**). All other registers are destroyed. The fork routine executes at the IPL indicated by the fork lock index stored in FKB$B_FLCK.

Description

EXE_STD$PRIMITIVE_FORK_WAIT moves the contents of **fr3** and **fr4** into FKB$Q_FR3 and FKB$Q_FR4 respectively. Having obtained the MEGA spin lock, it inserts the fork block indicated by **fkb** at end of the fork-and-wait queue (EXE$GL_FKWAITBL) and releases the spin lock.

Up to one second later, the software timer interrupt service routine will remove this and all other entries from the fork-and-wait queue and resume their respective fork routines.

Macro

FORK_WAIT

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$PRIMITIVE_FORK_WAIT is a call-based routine that performs the same operation as the JSB-based routine EXE$PRIMITIVE_FORK_WAIT on OpenVMS Alpha systems. The OpenVMS VAX routines EXE$FORK and EXE$IOFORK are not provided on OpenVMS Alpha systems.

EXE_STD$QIOACPPKT

Delivers an IRP to the appropriate ACP or XQP.

Module

SYSQIOREQ

Format

status = EXE_STD$QIOACPPKT (irp, pcb, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet.

pcb
Process control block.

ucb
Unit control block.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

EXE_STD$QIOACPPKT is called by, and returns to, a driver’s FDT routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

CALL_QIOACPPKT [do_ret=YES]

where:

do_ret indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

In an Alpha driver, the calls EXE_STD$QIOACPPKT using the current contents of R3, R4, and R5 as the **irp**, **pcb**, and **ucb** arguments, respectively. When EXE_STD$QIOACPPKT returns control to the code generated by a default invocation of CALL_QIOACPPKT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE$QIOACPPKT returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$QIOACPPKT returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status (SS$_NORMAL) in the FDT context structure. The $QIO system service retrieves the status from this structure.

EXE_STD$QIODRVPKT

Delivers an IRP to a driver's start-I/O routine or pending-I/O queue.

Module

SYSQIOREQ

Format

status = EXE_STD$QIODRVPKT (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet. EXE_STD$QIODRVPKT clears IRP$PS_FDT_CONTEXT. The caller of EXE_STD$QIODRVPKT should not access the IRP after the routine returns SS$_FDT_COMPL status.

ucb
Unit control block.

EXE_STD$QIODRVPKT (by means of the call to EXE_STD$INSIOQ) reads the following UCB fields:

Field	Contents
UCB$B_FLCK	Fork lock index
UCB$L_STS	UCB$V_BSY set if device is busy, clear if device is idle
UCB$L_IOQFL	Address of pending-I/O queue listhead
UCB$L_QLEN	Length of pending-I/O queue

EXE_STD$QIODRVPKT (by means of the call to EXE_STD$INSIOQ) writes the following UCB fields:

Field	Contents
UCB$L_STS	UCB$V_BSY set
UCB$W_QLEN	Increased

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

EXE_STD$QIODRVPKT is called by, and returns to, a driver's FDT routine at IPL$_ASTDEL.

Description

The FDT completion routine EXE_STD$QIODRVPKT delivers an IRP to the driver's start-I/O routine or pending-I/O queue.

EXE_STD$QIODRVPKT clears the pointer to the FDT context structure in IRP$PS_FDT_CONTEXT and calls. EXE_STD$INSIOQ checks the status of the device and calls either EXE_STD$INSERT_IRP or IOC_STD$INITIATE to place the IRP in the device's pending-I/O queue or deliver it to the driver's start-I/O routine, respectively.

When EXE_STD$INSIOQ returns, EXE_STD$QIODRVPKT returns SS$_FDT_COMPL status to its caller, passing SS$_NORMAL as the final $QIO completion status in the FDT context structure.

The image that requested the I/O operation receives SS$_NORMAL status, indicating that the I/O request has completed without device-independent error.

Macro

```
CALL_QIODRVPKT [do_ret=YES]
```

where:

> **do_ret** indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

In an Alpha driver, the CALL_QIODRVPKT macro clears IRP$PS_FDT_CONTEXT and calls EXE_STD$INSIOQ, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively. When EXE_STD$INSIOQ returns control to the code generated by a default invocation of $QIODRVPKT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The address of the PCB, supplied as input to EXE$QIODRVPKT on OpenVMS VAX systems, is not provided as input to EXE_STD$QIODRVPKT.

- Unlike EXE$QIODRVPKT, EXE_STD$QIODRVPKT does not lower IPL to 0 before exiting. EXE_STD$QIODRVPKT returns to its caller at the caller's IPL.
- EXE$QIODRVPKT returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$QIODRVPKT returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status (SS$_NORMAL) in the FDT context structure. The $QIO system service retrieves the status from this structure.

EXE_STD$QXQPPKT

Inserts an I/O request packet on the end of the XQP work queue and initiates its processing if it is the only request on the queue.

Module

SYSQIOREQ

Format

status = EXE_STD$QXQPPKT (pcb, acb)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required
acb	ACB	input	reference	required

pcb
Process control block.

acb
AST control block within the IRP.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

EXE_STD$QXQPPKT is called by, and returns to, a driver’s FDT routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

CALL_QXQPPKT

In an Alpha driver, the CALL_QXQPPKT macro calls EXE_STD$QXQPPKT using the current contents of R4 and R5 as the **pcb** and **acb** arguments, respectively. Status is returned in R0 and in the FDT_CONTEXT structure.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE$QXQPPKT returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$QXQPPKT returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status (SS$_NORMAL) in the FDT context structure. The $QIO system service retrieves the status from this structure.

EXE_STD$READ

Translates a logical read function into a physical read function, transfers $QIO system service parameters to the IRP, validates and prepares a user buffer, and aborts the request or proceeds with a direct-I/O, DMA write operation.

Module

SYSQIOFDT

Format

status = EXE_STD$READ (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request.

EXE_STD$READ reads the following IRP fields:

Field	Contents
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the buffer’s virtual address.
IRP$L_QIO_P2	$QIO system service **p2** argument, containing the number of bytes in transfer. The maximum number of bytes that EXE_STD$READ can transfer is 65,535 (128 pages minus one byte).
IRP$L_QIO_P4	$QIO system service **p4** argument, containing the carriage control byte.
IRP$L_FUNC	I/O function code.
IRP$B_RMOD	Access mode of the caller of the $QIO system service.

EXE_STD$READ writes the following IRP fields:

Field	Contents
IRP$B_CARCON	Carriage control byte (from IRP$L_QIO_P4)
IRP$L_FUNC	Logical read function code converted to physical
IRP$L_STS	IRP$V_FUNC set to indicate read function

Field	Contents
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow write access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	The I/O request has been successfully queued.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$READ as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$READ to prepare a direct-I/O read request. A driver cannot specify EXE_STD$READ for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their own upper-level FDT action routines to handle them.

EXE_STD$READ performs the following functions:

- Sets IRP$V_FUNC in IRP$L_STS to indicate a read function
- Copies the **p4** argument of the $QIO request from IRP$L_QIO_P4 to IRP$B_CARCON

- Translates a logical read function to a physical read function and stores the new function code in IRP$L_FUNC.
- Examines the size of the transfer, as specified in the **p2** argument of the $QIO request (IRP$L_QIO_P2), and takes one of the following actions:
  - If the transfer byte count is zero, EXE_STD$READ invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$READ regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

    The driver start-I/O routine should check for zero-length buffers to avoid mapping to adapter node space. An attempted mapping can cause a system failure.
  - If the byte count is not zero, EXE_STD$READ calls EXE_STD$READLOCK, specifying 0 as the **err_rout** argument.

EXE_STD$READLOCK invokes the $READCHK macro, which calls EXE_STD$READCHK.

EXE_STD$READCHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, it calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, EXE_STD$READCHK returns to EXE_STD$READLOCK with SS$_BADPARAM status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$READLOCK immediately returns to EXE_STD$READ, passing these status values. EXE_STD$READ, in turn, returns to the $QIO system service.
- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:
  - If the buffer allows write access, EXE_STD$READCHK sets IRP$V_FUNC in IRP$L_STS and returns SS$_NORMAL in R0 to EXE_STD$READLOCK.
  - If the buffer does not allow write access, EXE_STD$READCHK calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_ACCVIO. When it regains control, EXE_STD$READCHK returns to EXE_STD$READLOCK with SS$_ACCVIO status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$READLOCK immediately returns to EXE_STD$READ, passing these status values. EXE_STD$READ returns to the $QIO system service.

If EXE_STD$READCHK succeeds, EXE_STD$READLOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$READLOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$READLOCK. EXE_STD$READLOCK returns immediately to EXE_STD$READ, passing to it this status value.

EXE_STD$READ invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$READ regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$READLOCK.

  For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$READLOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$READLOCK returns EXE_STD$READ the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$READ returns to the $QIO system service.

  For page fault status, EXE_STD$READLOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine EXE$READ expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$READ.

- EXE$READ returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL, SS$_ACCVIO, or SS$_BADPARAM, or SS$_INSFWSL) in R0. EXE_STD$READ returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$READCHK

Verifies that a process has write access to the pages in the buffer specified in a $QIO request.

Module

SYSQIOFDT

Format

status = EXE_STD$READCHK (irp, pcb, ucb, buf, bufsiz)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
buf	address	input	reference	required
bufsiz	integer	input	value	required

irp
I/O request packet for the current I/O request.

EXE_STD$READCHK reads IRP$B_RMOD to determine the access mode of the caller of the $QIO system service.

EXE_STD$READCHK writes the following IRP fields:

Field	Contents
IRP$L_STS	IRP$V_FUNC set, indicating a read function
IRP$L_BCNT	Size of transfer in bytes

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

buf
Virtual address of buffer.

bufsiz
Number of bytes in transfer.

Return Values

SS$_NORMAL	The buffer is write-accessible.
SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow write access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Nothing has occurred yet to prevent the I/O request from being successfully queued. This is the initial value of the status field in an FDT_CONTEXT structure.

Context

The FDT support routine EXE_STD$READLOCK, or a driver-specific FDT routine, calls EXE_STD$READCHK at IPL$_ASTDEL.

Description

A driver FDT routine calls the system-supplied FDT support routine EXE_STD$READCHK to check the write accessibility of an I/O buffer supplied in a $QIO request for a read function.

EXE_STD$READCHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, it calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, EXE_STD$READCHK returns to its caller with SS$_BADPARAM status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.

- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:
  - If the buffer allows write access, EXE_STD$READCHK sets IRP$V_FUNC in IRP$L_STS and returns SS$_NORMAL in R0 to its caller.
  - If the buffer does not allow write access, EXE_STD$READCHK calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_ACCVIO. When it regains control, EXE_STD$READCHK returns to its caller with SS$_ACCVIO status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.

The caller of EXE_STD$READCHK must examine the status in R0:

- If the status is SS$_NORMAL, the buffer is write-accessible.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the I/O request to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS.

Certain drivers must perform additional processing to back out an I/O request after it has aborted. For instance, if the driver has locked multiple buffers into memory for a single I/O request, it must unlock them once the request has been aborted. A driver cannot access the IRP once it has received SS$_FDT_COMPL status. If you know you need access to information stored in the IRP to back out an I/O request that has been aborted, you must store that information elsewhere prior to calling EXE_STD$READCHK.

Macro

CALL_READCHK
CALL_READCHKR

In an Alpha driver, CALL_READCHK simulates a JSB to EXE$READCHK and CALL_READCHKR simulates a JSB to EXE$READCHKR. Both macros call EXE_STD$READCHK using the current contents of R3, R4, R5, R0, and R1 as the **irp**, **pcb**, **ucb**, **buf**, and **bufsiz** arguments, respectively.

When EXE_STD$READCHK returns, $READCHK and $READCHKR move 1 into R2 to indicate a read operation and examines the return status:

- If success status (SS$_NORMAL) is returned, CALL_READCHK and CALL_READCHKR copy the contents of IRP$L_BCNT into R1. CALL_READCHK writes the starting address of the I/O buffer in R0; CALL_READCHKR preserves the return status value in R0.
- If failure status (SS$_FDT_COMPL) is returned, CALL_READCHK returns to FDT dispatching code in the $QIO system service. CALL_READCHKR does not return control to $QIO.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$READCHK replaces EXE$READCHK and EXE$READCHKR. For compatibility with the VAX routines, use the CALL_READCHK and CALL_READCHKR macros.
- EXE$READCHK and EXE$READCHKR expect as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$READCHK.
- The order in which formal parameters are passed to EXE_STD$READCHK differs from the order in which they are provided in registers to the VAX routines EXE$READCHK and EXE$READCHKR.
- EXE$READCHK and EXE$READCHKR provide a mechanism by which a driver callback routine or coroutine obtains control upon an error condition prior to the abortion of an I/O request. The design of FDT processing for OpenVMS Alpha device drivers guarantees that the caller of EXE_STD$READCHK regains control whether the read check operation is successful. The caller must examine the return status in R0 (SS$_NORMAL indicates the buffer is write accessible, SS$_FDT_COMPL indicates that the operation failed and the request has been aborted) and take appropriate

action. Final $QIO completion status, indicating the reason the operation failed, is stored in the FDT_CONTEXT structure.

- Driver code that services failure status (SS$_FDT_COMPL) from EXE_STD$READLOCK (for instance a callback routine formerly specified to EXE$READLOCK_ERR) cannot access information in the IRP. If the driver anticipates handling failure status by using the contents of IRP fields, it must store these fields elsewhere before calling EXE_STD$READLOCK.

  This is especially important for driver code that expects EXE_STD$READCHK to access the transfer size in R1 after the call. Unlike EXE$READCHK and EXE$READCHKR, EXE_STD$READCHK does not preserve the contents of R1 and R3 across the call. If you must repeat a CALL_READCHK macro invocation, you must be sure to reload R0, R1, and R3 with the virtual address of the buffer, the transfer size, and the address of the IRP, respectively, before each subsequent invocation.

- Upon successful completion, EXE$READCHK and EXE$READCHKR set R2 to 1 for a read function. EXE_STD$READCHK does not provide R2 as output; a driver can determine whether a function is read or write by examining IRP$V_FUNC in IRP$L_STS.

EXE_STD$READLOCK

Validates and prepares a user buffer for a direct-I/O, DMA write operation.

Module

SYSQIOFDT

Format

status = EXE_STD$READLOCK (irp, pcb, ucb, ccb, buf, bufsiz, err_rout)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required
buf	address	input	reference	required
bufsiz	integer	input	value	required
err_rout	procedure value	input	value	required

irp
I/O request packet for the current I/O request.

EXE_STD$READLOCK reads IRP$B_RMOD to determine the access mode of the caller of the $QIO system service.

EXE_STD$READLOCK writes the following IRP fields:

Field	Contents
IRP$L_STS	IRP$V_FUNC set, indicating a read function
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

buf
Virtual address of buffer.

bufsiz
Number of bytes in transfer

err_rout
Procedure value of error-handling callback routine, or 0 if the driver does not process errors.

A driver typically specifies an error-handling callback routine when the driver must lock multiple areas into memory for a single I/O request and regain control to unlock these areas, if the request is to be aborted. The routine performs those tasks required before the request is backed out of or aborted. Such operations could include calling MMG_STD$UNLOCK to release previous buffers participating in the I/O operation. The error-handling routine must preserve R0 and R1 and return back to EXE_STD$READLOCK.

Chapter 8 describes the error-handling callback routine interface.

Return Values

SS$_NORMAL	The buffer is write-accessible and has been locked in memory.
SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow write access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Nothing has occurred yet to prevent the I/O request from being successfully queued. This is the initial value of the status field in an FDT_CONTEXT structure.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

The system-supplied upper-level FDT action routine EXE_STD$READ, or a driver-specific upper-level FDT action routine, calls EXE_STD$READLOCK at IPL$_ASTDEL.

Description

A driver FDT routine calls the system-supplied FDT support routine EXE_STD$READLOCK to check the write accessibility of an I/O buffer supplied in a $QIO request for a read function, and lock the buffer in memory in preparation for a DMA write operation.

A driver cannot specify EXE_STD$READ for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their own FDT routines to handle them.

EXE_STD$READLOCK invokes the $READCHK macro, which calls EXE_STD$READCHK.

EXE_STD$READCHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, EXE_STD$READCHK returns SS$_BADPARAM status to EXE_STD$READLOCK.

- Determines if the specified buffer is write accessible for a read I/O function, with one of the following results:
  - If the buffer allows write access, EXE_STD$READCHK sets IRP$V_FUNC in IRP$L_STS and returns SS$_NORMAL in R0 to EXE_STD$READLOCK.
  - If the buffer does not allow write access, EXE_STD$READCHK returns SS$_ACCVIO status to EXE$_STD$READLOCK.

If error status (SS$_BADPARAM or SS$_ACCVIO) is returned, EXE_STD$READLOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$READLOCK. When the callback routine returns (or if no callback routine is specified), EXE_STD$READLOCK calls EXE_STD$ABORTIO, passing it the error status as **qio_sts**. EXE_STD$ABORTIO returns to EXE_STD$READLOCK with the error status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$READLOCK immediately returns to its caller, passing these status values.

If SS$_NORMAL status is returned, EXE_STD$READLOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$READLOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$READLOCK. EXE_STD$READLOCK returns immediately to its caller, passing to it this status value.
- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$READLOCK. EXE_STD$READLOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$READLOCK. When

the callback routine returns (or if no callback routine is specified), EXE_STD$READLOCK proceeds as follows:

- For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$READLOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$READLOCK returns to its caller the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.
- For page fault status, EXE_STD$READLOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

The caller of EXE_STD$READLOCK must examine the status in R0:

- If the status is SS$_NORMAL, the buffer is write accessible and has been successfully locked into memory and the starting virtual address of the page table entries that map the buffer is available in IRP$L_SVAPTE.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the I/O request to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS. Ordinarily a driver specifies an error-handling callback routine to process such errors.

  Note that a driver cannot access the IRP once it has received SS$_FDT_COMPL status. If you know you need access to information stored in the IRP to back out an I/O request that has been aborted, you must store that information elsewhere prior to calling EXE_STD$READLOCK.

Macro

CALL_READLOCK
CALL_READLOCK_ERR [interface_warning=YES]

where:

interface_warning=**YES**, the default, specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the VAX version of the corresponding system routine. **interface_warning**=**NO** suppresses the warning.

In an Alpha driver, the CALL_READLOCK simulates a JSB to EXE$READLOCK and CALL_READLOCK_ERR simulates a JSB to EXE$READLOCK_ERR. CALL_READLOCK calls EXE_STD$READLOCK, specifying 0 as the **err_rout** argument; CALL_READLOCK_ERR also calls EXE_STD$READLOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsiz** arguments, respectively.

When EXE_STD$READLOCK or EXE_STD$READLOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro copies the contents of IRP$L_SVAPTE into R1 and writes a 1 to R2 to indicate a read operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro writes a 1 to R2 to indicate a read operation and returns to FDT dispatching code in the $QIO system service.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$READLOCK replaces EXE$READLOCK and EXE$READLOCK_ERR. For compatibility with the VAX routines, use the CALL_READLOCK and CALL_READLOCK_ERR macros.
- EXE$READLOCK and EXE$READLOCK_ERR expect as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$READLOCK.
- The order in which formal parameters are passed to EXE_STD$READLOCK differs from the order in which they are provided in registers to the VAX routines EXE$READLOCK and EXE$READLOCK_ERR.
- EXE$READLOCK_ERR provides a mechanism by which a driver callback routine obtains control upon an error condition prior to the abortion of an I/O request. EXE_STD$READLOCK accepts the address of an error-handling callback routine in the **err_rout** argument. The error-handling routine is called after an I/O request encounters a buffer access or memory allocation failure and before the request is aborted.
- The design of FDT processing for OpenVMS Alpha drivers guarantees that the caller of EXE_STD$READLOCK regains control whether the read lock operation is successful. When a driver regains control from a call to EXE_STD$READLOCK, return status in R0 indicates that the buffer has been successfully locked (SS$_NORMAL) or that the operation failed and the request has been aborted (SS$_FDT_COMPL). The driver must check the return status and take appropriate action. Final $QIO completion status, indicating the reason the operation failed, is stored in the FDT_CONTEXT structure.

  Normally, a driver services a read lock failure by supplying the address of an error-handling callback routine to EXE_STD$READLOCK.
- Driver code that executes after receiving failure status (SS$_FDT_COMPL) from EXE_STD$READLOCK cannot access information in the IRP. If the driver anticipates accessing IRP fields when EXE_STD$READLOCK returns, it must store these fields elsewhere before calling EXE_STD$READLOCK.
- Upon successful completion, EXE$READLOCK and EXE$READLOCK_ERR provide as output the system virtual address of the first process PTE that maps the buffer in R1 and in IRP$L_SVAPTE. Because EXE_STD$READLOCK does not provide R1 as output, a driver must obtain this information from IRP$L_SVAPTE. Similarly, the VAX routines set R2 to 1

for a read function and clear it otherwise. EXE_STD$READLOCK does not provide R2 as output; a driver can determine whether a function is read or write by examining IRP$V_FUNC in IRP$L_STS.

EXE_STD$SENSEMODE

Copies device-dependent characteristics from the device's UCB into the second longword of the I/O status block (IOSB) specified in a $QIO system service call, and completes the I/O operation successfully.

Module

SYSQIOFDT

Format

status = EXE_STD$SENSEMODE (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request. EXE_STD$SENSEMODE reads the device-dependent status stored in UCB$L_DEVDEPEND.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$SENSEMODE as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$SENSEMODE to process the sense-device-mode (IO$_SENSEMODE) and sense-device-characteristics (IO$_SENSECHAR) I/O functions.

EXE_STD$SENSEMODE loads the contents of UCB$L_DEVDEPEND into the second longword of the I/O status block (IOSB) specified in the original $QIO system service call. It then places SS$_NORMAL status into the FDT_CONTEXT structure and transfers control to EXE_STD$FINISHIO to insert the IRP in the local processor's I/O postprocessing queue.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine EXE$SENSEMODE (used by VAX drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$SENSEMODE.
- EXE$SENSEMODE returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$SENSEMODE returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$SETCHAR, EXE_STD$SETMODE

Write device-specific status and control information into the device's UCB and complete the I/O request (EXE_STD$SETCHAR); or write the information into the IRP and deliver the IRP to the driver's start-I/O routine (EXE_STD$SETMODE).

Module

SYSQIOFDT

Format

status = EXE_STD$SETCHAR (irp, pcb, ucb, ccb)

status = EXE_STD$SETMODE (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request.

EXE_STD$SETCHAR and EXE_STD$SETMODE read the following IRP fields:

Field	Contents
IRP$L_FUNC	I/O function code supplied in the $QIO request
IRP$B_RMOD	Mode of the $QIO caller
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the device characteristics quadword.

EXE_STD$SETMODE writes the following IRP fields:

Field	Contents
IRP$L_MEDIA	First longword of device characteristics
IRP$L_MEDIA+4	Second longword of device characteristics

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

EXE_STD$SETCHAR writes the following UCB fields:

Field	Contents
UCB$B_DEVCLASS	Byte 0 of device characteristics quadword
UCB$B_DEVTYPE	Byte 1 of device characteristics quadword
UCB$W_DEVBUFSIZ	Bytes 2 and 3 of device characteristics quadword
UCB$L_DEVDEPEND	Bytes 4 through 7 of device characteristics quadword

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.
SS$_ACCVIO	Process calling the $QIO system service with the IO$_SETMODE or IO$_SETCHAR function does not have read access to the quadword containing the new device characteristics.
SS$_ILLIOFUNC	IO$_SETMODE and IO$_SETCHAR functions are not legal for disk devices.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$SETCHAR and EXE_STD$SETMODE as upper-level FDT action routines at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$SETCHAR or EXE_STD$SETMODE to process the set-device-mode (IO$_SETMODE) and set-device-characteristics (IO$_SETCHAR) functions, respectively. If setting device characteristics requires device activity or synchronization with fork processing, the driver's FDT_ACT macro invocation *must* specify EXE_STD$SETMODE. Otherwise, it can specify EXE_STD$SETCHAR.

EXE_STD$SETCHAR and EXE_STD$SETMODE examine the current value of UCB$B_DEVCLASS to determine whether the device permits the specified function. If the device class is disk (DC$_DISK), the routines place SS$_ILLIOFUNC status in the FDT_CONTEXT structure and transfer control to EXE_STD$ABORTIO to terminate the request.

EXE_STD$SETCHAR and EXE_STD$SETMODE then ensure that the process has read access to the quadword containing the new device characteristics. If it does not, the routines place SS$_ACCVIO status in the FDT_CONTEXT structure and transfer control to EXE_STD$ABORTIO to terminate the request.

If the request passes these checks, EXE_STD$SETCHAR and EXE_STD$SETMODE proceed as follows:

- EXE_STD$SETCHAR stores the specified characteristics in the UCB. For an IO$_SETCHAR function, the device type and class fields (UCB$B_DEVCLASS and UCB$B_DEVTYPE, respectively) receive the first word of data. For both IO$_SETCHAR and IO$_SETMODE functions, EXE_STD$SETCHAR writes the second word into the default-buffer-size field (UCB$W_DEVBUFSIZ) and the third and fourth words into the device-dependent-characteristics field (UCB$Q_DEVDEPEND).

  Finally, EXE_STD$SETCHAR stores normal completion status (SS$_NORMAL) in the FDT_CONTEXT structure and transfers control to the FDT completion routine EXE_STD$FINISHIO to insert the IRP in the local processor's I/O postprocessing queue. EXE_STD$FINISHIO returns to EXE_STD$SETCHAR with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.
- EXE_STD$SETMODE stores the specified quadword of characteristics in IRP$L_MEDIA, places normal completion status (SS$_NORMAL) in the FDT_CONTEXT structure, and transfers control to FDT completion routine EXE_STD$QIODRVPKT to deliver the IRP to the driver's start-I/O routine. EXE_STD$QIODRVPKT returns to EXE_STD$SETMODE with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.

The driver's start-I/O routine copies data from IRP$L_MEDIA and the following longword into UCB$W_DEVBUFSIZ, UCB$L_DEVDEPEND, and, if the I/O function is IO$_SETCHAR, UCB$B_DEVCLASS and UCB$B_DEVTYPE as well.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routines EXE$SETCHAR and EXE$SETMODE (used by OpenVMS VAX device drivers) expect as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$SETCHAR and EXE_STD$SETMODE.
- EXE$SETCHAR and EXE$SETMODE return control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL, SS$_ACCVIO, or SS$_ILLIOFUNC) in R0. EXE_STD$SETCHAR or EXE_STD$SETMODE returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$SNDEVMSG

Builds and sends a device-specific message to the mailbox of a system process, such as the job controller or OPCOM.

Module

MBDRIVER

Format

status = EXE_STD$SNDEVMSG (mb_ucb, msgtyp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
mb_ucb	MB_UCB	input	reference	required
msgtyp	integer	input	value	required
ucb	UCB	input	reference	required

mb_ucb
Mailbox UCB. (SYS$AR_JOBCTLMB contains the address of the job controller's mailbox; SYS$AR_OPRMBX contains the address of OPCOM's mailbox.)

msgtyp
Message type. OPCOM message types have the prefix OPC$_ and are defined by the $OPCMSG macro in SYS$LIBRARY:STARLET.MLB.

ucb
Device UCB. EXE_STD$SNDEVMSG reads the following UCB fields:

UCB$W_UNIT	Device unit number.
UCB$L_DDB	Address of device DDB. EXE_STD$SNDEVMSG constructs the device controller name from DDB$T_NAME and mailbox UCB fields.

Return Values

SS$_DEVNOTMBX	**mb_ucb** does not specify a mailbox UCB.
SS$_INSFMEM	The system is unable to allocate memory for the message.
SS$_MBFULL	The message mailbox is full of messages.
SS$_MBTOOSML	The message is too large for the mailbox.
SS$_NOPRIV	The caller lacks privilege to write to the mailbox.
SS$_NORMAL	Normal, successful completion.

Context

Because EXE_STD$SNDEVMSG raises IPL to IPL$_MAILBOX and obtains the MAILBOX spin lock in a multiprocessing environment, its caller cannot be executing above IPL$_MAILBOX. EXE_STD$SNDEVMSG returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

EXE_STD$SNDEVMSG builds a 32-byte message on the stack that includes the following information:

Bytes	Contents
0 and 1	Low word of **msgtyp** parameter
2 and 3	Device unit number (UCB$W_UNIT)
4 through 31	Counted string of device controller name, formatted as *node$controller* for clusterwide devices

EXE_STD$SNDEVMSG then calls EXE_STD$WRTMAILBOX to send the message to a mailbox.

Macro

CALL_SNDEVMSG [save_r1]

where:

> **save_r1** indicates that the macro should preserve register R1 across the call to COM_STD$POST. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

In an Alpha driver, the CALL_SNDEVMSG macro calls EXE_STD$SNDEVMSG, using the current contents of R3, R4, and R5 as the **mb_ucb**, **msgtyp**, and **ucb** arguments, respectively. It returns status in R0. Unless you specify **save_r1**=**NO**, the macro preserves R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$SNDEVMSG replaces EXE$SNDEVMSG (used by OpenVMS VAX drivers). Unlike EXE$SNDEVMSG, EXE_STD$SNDEVMSG does not preserve R1 across the call.

EXE_STD$WRITE

Translates a logical write function into a physical write function, transfers $QIO system service parameters to the IRP, validates and prepares a user buffer, and aborts the request or proceeds with a direct-I/O, DMA read operation.

Module

SYSQIOFDT

Format

status = EXE_STD$WRITE (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request.

EXE_STD$WRITE reads the following IRP fields:

Field	Contents
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the buffer's virtual address.
IRP$L_QIO_P2	$QIO system service **p2** argument, containing the number of bytes in transfer. The maximum number of bytes that EXE_STD$WRITE can transfer is 65,535 (128 pages minus one byte).
IRP$L_QIO_P4	$QIO system service **p4** argument, containing the carriage control byte.
IRP$L_FUNC	I/O function code.
IRP$B_RMOD	Access mode of the caller of the $QIO system service.

EXE_STD$WRITE writes the following IRP fields:

Field	Contents
IRP$B_CARCON	Carriage control byte (from IRP$L_QIO_P4)
IRP$L_FUNC	Logical write function code converted to physical

Field	Contents
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow read access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	The I/O request has been successfully queued.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$WRITE as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$WRITE to prepare a direct-I/O write request. A driver cannot specify EXE_STD$WRITE for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their own upper-level FDT action routines to handle them.

EXE_STD$WRITE performs the following functions:

- Copies the **p4** argument of the $QIO request from IRP$L_QIO_P4 to IRP$B_CARCON
- Translates a logical write function to a physical write function and stores the new function code in IRP$L_FUNC.

- Examines the size of the transfer, as specified in the **p2** argument of the $QIO request (IRP$L_QIO_P2), and takes one of the following actions:
    - If the transfer byte count is zero, EXE_STD$WRITE invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$WRITE regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

      The driver start-I/O routine should check for zero-length buffers to avoid mapping to adapter node space. An attempted mapping can cause a system failure.
    - If the byte count is not zero, EXE_STD$WRITE calls EXE_STD$WRITELOCK, passing 0 as the value of the **err_rout** argument.

EXE_STD$WRITELOCK invokes the $WRITECHK macro, which calls EXE_STD$WRITECHK.

EXE_STD$WRITECHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, it calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, EXE_STD$WRITECHK returns to EXE_STD$WRITELOCK with SS$_BADPARAM status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$WRITELOCK immediately returns to EXE_STD$WRITE, passing these status values. EXE_STD$WRITE, in turn, returns to the $QIO system service.
- Determines if the specified buffer is read accessible for a write I/O function, with one of the following results:
    - If the buffer allows read access returns SS$_NORMAL in R0 to EXE_STD$WRITELOCK.
    - If the buffer does not allow read access, EXE_STD$WRITECHK calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_ACCVIO. When it regains control, EXE_STD$WRITECHK returns to EXE_STD$WRITELOCK with SS$_ACCVIO status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$WRITELOCK immediately returns to EXE_STD$WRITE, passing these status values. EXE_STD$WRITE returns to the $QIO system service.

If EXE_STD$WRITECHK succeeds, EXE_STD$WRITELOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$WRITELOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$WRITELOCK. EXE_STD$WRITELOCK returns immediately to EXE_STD$WRITE, passing to it this status value.

EXE_STD$WRITE invokes the $QIODRVPKT macro to deliver the IRP to the driver's start-I/O routine. EXE_STD$WRITE regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure. It returns to the $QIO system service, passing these status values.

- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$WRITELOCK.

  For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$WRITELOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$WRITELOCK returns EXE_STD$WRITE the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$WRITE returns to the $QIO system service.

  For page fault status, EXE_STD$WRITELOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine EXE$WRITE (used by OpenVMS VAX device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$WRITE.
- EXE$WRITE returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL, SS$_ACCVIO, or SS$_BADPARAM, or SS$_INSFWSL) in R0. EXE_STD$WRITE returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

EXE_STD$WRITECHK

Verifies that a process has read access to the pages in the buffer specified in a $QIO request.

Module

SYSQIOFDT

Format

status = EXE_STD$WRITECHK (irp, pcb, ucb, buf, bufsiz)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
buf	address	input	reference	required
bufsiz	integer	input	value	required

irp
I/O request packet for the current I/O request.

EXE_STD$WRITECHK reads IRP$B_RMOD to determine the access mode of the caller of the $QIO system service.

EXE_STD$WRITECHK writes the size of the transfer in bytes to IRP$L_BCNT.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

buf
Virtual address of buffer.

bufsiz
Number of bytes in transfer.

Return Values

SS$_NORMAL	The buffer is read-accessible.
SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow read access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Nothing has occurred yet to prevent the I/O request from being successfully queued. This is the initial value of the status field in an FDT_CONTEXT structure.

Context

The FDT support routine EXE_STD$WRITELOCK, or a driver-specific FDT routine, calls EXE_STD$WRITECHK at IPL$_ASTDEL.

Description

A driver FDT routine calls the system-supplied FDT support routine EXE_STD$WRITECHK to check the read accessibility of an I/O buffer supplied in a $QIO request for a write function.

EXE_STD$WRITECHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, it calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_BADPARAM. When it regains control, EXE_STD$WRITECHK returns to its caller with SS$_BADPARAM status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.
- Determines if the specified buffer is read accessible for a write I/O function, with one of the following results:
  - If the buffer allows read access, EXE_STD$WRITECHK returns SS$_NORMAL in R0 to its caller.
  - If the buffer does not allow read access, EXE_STD$WRITECHK calls EXE_STD$ABORTIO, passing it a **qio_sts** of SS$_ACCVIO. When it regains control, EXE_STD$WRITECHK returns to its caller with SS$_ACCVIO status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.

The caller of EXE_STD$WRITECHK must examine the status in R0:

- If the status is SS$_NORMAL, the buffer is read-accessible.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the I/O request to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS.

Certain drivers must perform additional processing to back out an I/O request after it has aborted. For instance, if the driver has locked multiple buffers into memory for a single I/O request, it must unlock them once the request has been aborted. Note that a driver cannot access the IRP once it has received SS$_FDT_COMPL status. If you know you need access to information stored in the IRP to back out an I/O request that has been aborted, you must store that information elsewhere prior to calling EXE_STD$WRITELOCK.

Macro

CALL_WRITECHK
CALL_WRITECHKR

In an Alpha driver, CALL_WRITECHK simulates a JSB to EXE$WRITECHK and CALL_READCHKR simulates a JSB to EXE$READCHKR. Both macros call EXE_STD$READCHK using the current contents of R3, R4, R5, R0, and R1 as the **irp**, **pcb**, **ucb**, **buf**, and **bufsiz** arguments, respectively.

When EXE_STD$WRITECHK returns, CALL_WRITECHK and CALL_WRITECHKR clear R2 to indicate a write operation and examines the return status:

- If success status (SS$_NORMAL) is returned, CALL_WRITECHK and CALL_WRITECHKR copy the contents of IRP$L_BCNT into R1. CALL_WRITECHK writes the starting address of the I/O buffer in R0; CALL_WRITECHKR preserves the return status value in R0.
- If failure status (SS$_FDT_COMPL) is returned, CALL_WRITECHK returns to FDT dispatching code in the $QIO system service. CALL_WRITECHKR does not return control to $QIO.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$WRITECHK replaces EXE$WRITECHK and EXE$WRITECHKR (used by VAX drivers). For compatibility with the VAX routines, use the CALL_WRITECHK and CALL_WRITECHKR macros.
- EXE$WRITECHK and EXE$WRITECHKR expect as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$WRITECHK.
- The order in which formal parameters are passed to EXE_STD$WRITECHK differs from the order in which they are provided in registers to the EXE$WRITECHK and EXE$WRITECHKR routines .
- EXE$WRITECHK and EXE$WRITECHKR provide a mechanism by which a driver callback routine or coroutine obtains control upon an error condition prior to the abortion of an I/O request. The design of FDT processing for OpenVMS Alpha device drivers guarantees that the caller of EXE_STD$WRITECHK regains control whether the write check operation is successful. The caller must examine the return status in R0 (SS$_NORMAL indicates the buffer is read accessible, SS$_FDT_COMPL indicates that the operation failed and the request has been aborted) and take appropriate

action. Final $QIO completion status, indicating the reason the operation failed, is stored in the FDT_CONTEXT structure.

- Driver code that services failure status (SS$_FDT_COMPL) from EXE_STD$WRITELOCK (for instance, a callback routine formerly specified to EXE$WRITELOCK_ERR) cannot access information in the IRP. If the driver anticipates handling failure status by using the contents of IRP fields, it must store these fields elsewhere before calling EXE_STD$WRITELOCK.

  This is especially important for driver code that expects EXE_STD$WRITECHK to access the transfer size in R1 after the call. Unlike EXE$WRITECHK and EXE$WRITECHKR, EXE_STD$WRITECHK does not preserve the contents of R1 and R3 across the call. If you must repeat a CALL_WRITECHK macro invocation, be sure to reload R0, R1, and R3 with the virtual address of the buffer, the transfer size, and the address of the IRP, respectively, before each subsequent invocation.

- Upon successful completion, EXE$WRITECHK and EXE$WRITECHKR clear R2 to indicate a write function. EXE_STD$WRITECHK does not provide R2 as output; a driver can determine whether a function is write or read by examining IRP$V_FUNC in IRP$L_STS.

EXE_STD$WRITELOCK

Validates and prepares a user buffer for a direct-I/O, DMA read operation.

Module

SYSQIOFDT

Format

status = EXE_STD$WRITELOCK (irp, pcb, ucb, ccb, buf, bufsiz, err_rout)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required
buf	address	input	reference	required
bufsiz	integer	input	value	required
err_rout	procedure value	input	value	required

irp

I/O request packet for the current I/O request.

EXE_STD$WRITELOCK reads IRP$B_RMOD to determine the access mode of the caller of the $QIO system service.

EXE_STD$WRITELOCK writes the following IRP fields:

Field	Contents
IRP$L_SVAPTE	System virtual address of the PTE that maps the first page of the buffer
IRP$L_BOFF	Byte offset to start of transfer in page
IRP$L_OBOFF	Original byte offset into the first page of a segmented direct-I/O transfer
IRP$L_BCNT	Size of transfer in bytes

pcb

Process control block of the current process.

ucb

Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

buf
Virtual address of buffer.

bufsiz
Number of bytes in transfer.

err_rout
Procedure value of error-handling callback routine, or 0 if the driver does not process errors.

A driver typically specifies an error-handling callback routine when it must lock multiple areas into memory for a single I/O request and must regain control to unlock these areas, if the request is to be aborted. The routine performs those tasks required before the request is backed out of or aborted. Such operations could include calling MMG_STD$UNLOCK to release previous buffers participating in the I/O operation. The error-handling routine must preserve R0 and R1 and return back to EXE_STD$WRITELOCK.

Chapter 8 describes the error-handling callback routine interface.

Return Values

SS$_NORMAL	The buffer is read-accessible and has been locked in memory.
SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Buffer specified in **buf** parameter does not allow read access.
SS$_BADPARAM	**bufsiz** parameter is less than zero.
SS$_INSFWSL	Insufficient working set limit.
SS$_NORMAL	Nothing has occurred yet to prevent the I/O request from being successfully queued. This is the initial value of the status field in an FDT_ CONTEXT structure.
SS$_INSFWSL	Insufficient working set limit.
SS$_QIO_CROCK	Buffer page must be faulted into memory.

Context

The system-supplied upper-level FDT action routine EXE_STD$WRITE, or a driver-specific upper-level FDT action routine, calls EXE_STD$WRITELOCK at IPL$_ASTDEL.

Description

A driver FDT routine calls the system-supplied FDT support routine EXE_STD$WRITELOCK to check the read accessibility of an I/O buffer supplied in a $QIO request for a write function, and lock the buffer in memory in preparation for a DMA write operation.

A driver cannot specify EXE_STD$WRITE for buffered-I/O functions. Drivers that process functions that require an intermediate system buffer typically supply their FDT routines to handle them.

EXE_STD$WRITELOCK invokes the $WRITECHK macro, which calls EXE_STD$WRITECHK.

EXE_STD$WRITECHK performs the following actions:

- Moves the transfer byte count (**bufsiz** parameter) into IRP$L_BCNT.

  If the byte count is negative, EXE_STD$WRITECHK returns SS$_BADPARAM status to EXE_STD$READLOCK.

- Determines if the specified buffer is read accessible for a write I/O function, with one of the following results:
  - If the buffer allows read access, EXE_STD$WRITECHK returns SS$_NORMAL in R0 to EXE_STD$WRITELOCK.
  - If the buffer does not allow write access, EXE_STD$READCHK returns SS$_ACCVIO status to EXE$_STD$READLOCK.

If error status (SS$_BADPARAM or SS$_ACCVIO) is returned, EXE_STD$WRITELOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$WRITELOCK. When the callback routine returns (or if no callback routine is specified), EXE_STD$WRITELOCK calls EXE_STD$ABORTIO, passing it the error status as **qio_sts**. EXE_STD$ABORTIO returns to EXE_STD$WRITELOCK with the error status in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0. EXE_STD$WRITELOCK immediately returns to its caller, passing these status values.

If SS$_NORMAL status is returned, EXE_STD$WRITELOCK moves into IRP$L_BOFF and IRP$L_OBOFF the byte offset to the start of the buffer and calls MMG_STD$IOLOCK.

MMG_STD$IOLOCK attempts to lock into memory those pages that contain the buffer, with one of the following results:

- If MMG_STD$IOLOCK succeeds, EXE_STD$WRITELOCK stores in IRP$L_SVAPTE the system virtual address of the process PTE that maps the first page of the buffer, and returns SS$_NORMAL status in R0 to EXE_STD$WRITELOCK. EXE_STD$WRITELOCK returns immediately to its caller, passing to it this status value.
- If MMG_STD$IOLOCK fails, it returns SS$_ACCVIO, SS$_INSFWSL, or page fault status to EXE_STD$WRITELOCK. EXE_STD$WRITELOCK immediately calls the specified error-handling callback routine, passing to it the IRP, PCB, UCB, CCB, and status value. The callback routine must preserve R0 and R1 and return control to EXE_STD$WRITELOCK. When

the callback routine returns (or if no callback routine is specified), EXE_STD$WRITELOCK proceeds as follows:

- For SS$_ACCVIO and SS$_INSFWSL status, EXE_STD$WRITELOCK calls EXE_STD$ABORTIO, passing it one of these status values as a **qio_sts** argument. When it regains control, EXE_STD$WRITELOCK returns to its caller the specified status value in the FDT_CONTEXT structure and SS$_FDT_COMPL status in R0.
- For page fault status, EXE_STD$WRITELOCK sets the final $QIO status in the FDT_CONTEXT structure to SS$_QIO_CROCK and initializes FDT_CONTEXT$L_QIO_R1_VALUE to the virtual address to be faulted. It then adjusts the direct I/O count and AST count to the values they held before the I/O request, deallocates the IRP, and restarts the I/O request at the $QIO system service. This procedure is carried out so that the user process can receive ASTs while it waits for the page fault to complete. Once the page is faulted into memory, the $QIO system service will resubmit the I/O request.

The caller of EXE_STD$WRITELOCK must examine the status in R0:

- If the status is SS$_NORMAL, the buffer is write accessible and has been successfully locked into memory and the starting virtual address of the page table entries that map the buffer is available in IRP$L_SVAPTE.
- If the status is SS$_FDT_COMPL, an error has occurred that has caused the I/O request to be aborted. You can determine the reason for the failure from FDT_CONTEXT$L_QIO_STATUS. Ordinarily a driver specifies an error-handling callback routine to process such errors.

  Note that a driver cannot access the IRP once it has received SS$_FDT_COMPL status. If you know you need access to information stored in the IRP to back out an I/O request that has been aborted, you must store that information elsewhere prior to calling EXE_STD$WRITELOCK.

Macro

CALL_WRITELOCK
CALL_WRITELOCK_ERR [interface_warning=YES]

where:

> **interface_warning**=**YES**, the default, specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the VAX version of the corresponding system routine. **interface_warning**=**NO** suppresses the warning.

In an OpenVMS Alpha driver, the CALL_WRITELOCK simulates a JSB to EXE$WRITELOCK and CALL_WRITELOCK_ERR simulates a JSB to EXE$WRITELOCK_ERR. CALL_WRITELOCK calls EXE_STD$WRITELOCK, specifying 0 as the **err_rout** argument; CALL_WRITELOCK_ERR also calls EXE_STD$WRITELOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsiz** arguments, respectively.

When EXE_STD$WRITELOCK or EXE_STD$WRITELOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro moves the contents of IRP$L_SVAPTE into R1 and clears R2 to indicate a write operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro clears R2 to indicate a write operation and returns to FDT dispatching code in the $QIO system service.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$WRITELOCK replaces EXE$WRITELOCK and EXE$WRITELOCK_ERR. For compatibility with the VAX routines, use the CALL_WRITELOCK and CALL_WRITELOCK_ERR macros.
- EXE$WRITELOCK and EXE$WRITELOCK_ERR expect as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$WRITELOCK.
- The order in which formal parameters are passed to EXE_STD$WRITELOCK differs from the order in which they are provided in registers to the VAX routines EXE$WRITELOCK and EXE$WRITELOCK_ERR.
- EXE$WRITELOCK_ERR provides a mechanism by which a driver callback routine or coroutine obtains control upon an error condition prior to the abortion of an I/O request. EXE_STD$WRITELOCK accepts the address of an error-handling callback routine in the **err_rout** argument. The error-handling routine is called after an I/O request encounters a buffer access or memory allocation failure and before the request is aborted.
- The design of FDT processing for OpenVMS Alpha device drivers guarantees that the caller of EXE_STD$WRITELOCK regains control whether the write lock operation is successful. When a driver regains control from a call to EXE_STD$WRITELOCK, return status in R0 indicates that the buffer has been successfully locked (SS$_NORMAL) or that the operation failed and the request has been aborted (SS$_FDT_COMPL). The driver must check the return status and take appropriate action. Final $QIO completion status, indicating the reason the operation failed, is stored in the FDT_CONTEXT structure.

  Normally, a driver services a read lock failure by supplying the address of an error-handling callback routine to EXE_STD$WRITELOCK.
- Driver code that executes after receiving failure status (SS$_FDT_COMPL) from EXE_STD$WRITELOCK cannot access information in the IRP. If the driver anticipates accessing IRP fields when EXE_STD$WRITELOCK returns, it must store these fields elsewhere before calling EXE_STD$WRITELOCK.
- Upon successful completion, EXE$WRITELOCK and EXE$WRITELOCK_ERR provide as output the system virtual address of the first process PTE that maps the buffer in R1 and in IRP$L_SVAPTE. Because EXE_STD$WRITELOCK does not provide R1 as output, a driver must obtain this

information from IRP$L_SVAPTE. Similarly, the VAX routines clear R2 for a write function. EXE_STD$WRITELOCK does not provide R2 as output; a driver can determine whether a function is write or read by examining IRP$V_FUNC in IRP$L_STS.

EXE_STD$WRTMAILBOX

Sends a message to a mailbox.

Module

MBDRIVER

Format

status = EXE_STD$WRTMAILBOX (mb_ucb, msgsiz, msg)

Arguments

Argument	Type	Access	Mechanism	Status
mb_ucb	MB_UCB	input	reference	required
msgsiz	integer	input	value	required
msg	address	input	reference	required

mb_ucb
Mailbox UCB. (SYS$AR_JOBCTLMB contains the address of the job controller's mailbox; SYS$AR_OPRMBX contains the address of OPCOM's mailbox.)

msgsiz
Message size.

msg
Address of buffer containing the message.

Return Values

SS$_INSFMEM	The system is unable to allocate memory for the message.
SS$_MBFULL	The message mailbox is full of messages.
SS$_MBTOOSML	The message is too large for the mailbox.
SS$_NOPRIV	The caller lacks privilege to write to the mailbox.
SS$_NORMAL	Normal, successful completion.

Context

Because EXE_STD$WRTMAILBOX raises IPL to IPL$_MAILBOX and obtains the MAILBOX spin lock in a multiprocessing environment, its caller cannot be executing above IPL$_MAILBOX. EXE_STD$WRTMAILBOX returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

EXE_STD$WRTMAILBOX checks fields in the mailbox UCB (UCBW_MSGQUO, UCBW_DEVMSGSIZ) to determine whether it can deliver a message of the specified size to the mailbox. It also checks fields in the associated ORB to determine whether the caller is sufficiently privileged to write to the mailbox. Finally, it calls EXE$ALONONPAGED to allocate a block of nonpaged pool to contain the message. If it fails any of these operations, EXE_STD$WRTMAILBOX returns error status to its caller.

If it is successful thus far, EXE_STD$WRTMAILBOX creates a message and delivers it to the mailbox's message queue, adjusts its UCB fields accordingly, and returns success status to its caller.

Macro

CALL_WRTMAILBOX [save_r1]

where:

> **save_r1** indicates that the macro should preserve register R1 across the call to COM_STD$POST. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

In an Alpha driver, the CALL_WRTMAILBOX macro simulates a JSB to EXE$WRTMAILBOX in a VAX driver. CALL_WRTMAILBOX calls EXE_STD$WRTMAILBOX, using the current contents of R5, R3, and R4 as the **mb_ucb**, **msgsiz**, and **msg** arguments, respectively. It returns status in R0. Unless you specify **save_r1**=**NO**, the macro preserves the R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- EXE_STD$WRTMAILBOX replaces EXE$WRTMAILBOX (used by VAX drivers). The order in which formal parameters are passed to EXE_STD$WRTMAILBOX differs from the order in which they are provided in registers to the VAX routine EXE$WRTMAILBOX.
- Unlike EXE$WRTMAILBOX, EXE_STD$WRTMAILBOX does not preserve R1 across the call.

EXE_STD$ZEROPARM

Delivers an I/O request that requires no parameters to a driver's start-I/O routine.

Module

SYSQIOFDT

Format

status = EXE_STD$ZEROPARM (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request. EXE_STD$ZEROPARM clears IRP$L_MEDIA.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

FDT dispatching code in the $QIO system service calls EXE_STD$ZEROPARM as an upper-level FDT action routine at IPL$_ASTDEL.

Description

A driver specifies the system-supplied upper-level FDT action routine EXE_STD$ZEROPARM to process an I/O function code that has no required parameters.

EXE_STD$ZEROPARM clears IRP$L_MEDIA and invokes the $QIODRVPKT macro to deliver the IRP to the driver. EXE_STD$ZEROPARM regains control with SS$_FDT_COMPL status in R0 and a final $QIO system service status of SS$_NORMAL in the FDT_CONTEXT structure.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine EXE$ZEROPARM (used by OpenVMS VAX device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to EXE_STD$ZEROPARM.
- EXE$ZEROPARM returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. EXE_STD$ZEROPARM returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

IOC$ALOALTMAP, IOC$ALOALTMAPN, IOC$ALOALTMAPSP

Allocate a set of Q22–bus alternate map registers.

Notes for Converting VAX Drivers

Not supported on OpenVMS Alpha systems. 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.

Notes for Converting VAX Drivers

Not supported on OpenVMS Alpha systems. See the description of IOC$ALLOC_CNT_RES.

IOC$ALLOC_CNT_RES

Allocates the requested number of items of a counted resource.

Module

ALLOC_CNT_RES

Format

IOC$ALLOC_CNT_RES crab ,crctx

Context

IOC$ALLOC_CNT_RES conforms to the OpenVMS Alpha calling standard. Its caller must be executing at fork IPL, holding the corresponding fork lock.

Arguments

crab

VMS Usage:	address
type:	longword (signed)
access:	read only
mechanism:	by reference

Address of CRAB that describes the counted resource. For adapters that supply a counted resource, such as map registers, ADP$L_CRAB often contains this address.

crctx

VMS Usage:	address
type:	longword (signed)
access:	read only
mechanism:	by reference

Address of CRCTX structure that describes the request for the counted resource.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.

SS$_BADPARAM	Request count was greater than the total number of items managed by the CRAB or the total number of items defined by a bounded request. This status is also returned if the lower bound of the request (CRCTX$L_LOW_BOUND) is greater than the upper bound (CRCTX$L_UP_BOUND).
SS$_INSFMAPREG	Insufficient resources to satisfy request, or other requests precede this one in the resource-wait queue.

Description

IOC$ALLOC_CNT_RES allocates a requested number of items from a counted resource. 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 submitting it in a call to IOC$ALLOC_CNT_RES.

Field	Description
CRCTX$L_ITEM_CNT	Number of items to be allocated. When requesting map registers, this value in this field should include an extra map register to be allocated and loaded as a guard page to prevent runaway transfers.
CRCTX$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 queueing 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 performs the following tasks:

- It acquires the spin lock indicated by CRAB$L_SPINLOCK, raising IPL to IPL$_SYNCH in the process.
- If there are no waiters for the counted resource (that is, the resource wait queue headed by CRAB$L_WQFL is empty) or if the CRCTX describes a high-priority allocation request (CRCTX$V_HIGH_PRIO in CRCTX$L_FLAGS is set), IOC$ALLOC_CNT_RES attempts the allocation immediately. It scans the CRAB allocation array for a descriptor that contains as many free items as requested by the caller (in CRCTX$L_ITEM_CNT).

  In performing the scan, IOC$ALLOC_CNT_RES considers any indicated range of counted resource items that are to be involved in the scan, and limits its search to those item descriptors in the allocation array that describe items within these bounds. A bounded search is indicated by nonzero values in CRCTX$L_UP_BOUND and CRCTX$L_LOW_BOUND. IOC$ALLOC_CNT_RES rounds up the allocation request to the minimal allocation granularity, as indicated by CRAB$L_ALLOC_GRAN_MASK.

The number of the first resource item granted to the caller is placed in CRCTX$L_ITEM_NUM and CRCTX$V_ITEM_VALID is set in CRCTX$L_FLAGS.

- If this allocation attempt fails, 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.

When a counted resource deallocation occurs, the CRCTX is removed from the wait queue and the allocation is attempted again.

When the allocation succeeds, IOC$ALLOC_CNT_RES issues a JSB instruction to the callback routine (CRCTX$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 (CTCTX$Q_FR4)
R5	Contents of R5 at the time of the original allocation request (CRCTX$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, it typically proceeds to loads the map registers that have been allocated. It must preserve all registers it uses other than R0 through R5 and exit with an RSB instruction.

- It releases the spin lock indicated by CRAB$L_SPINLOCK (upon the condition that its caller did not already own that spin lock at the time of the call) and returns to its caller.

OpenVMS Alpha 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 uses 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 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.

IOC$ALLOC_CRAB

Allocates and initializes a counted resource allocation block (CRAB).

Module

ALLOC_CNT_RES

Format

IOC$ALLOC_CRAB item_cnt ,req_alloc_gran ,crab_ref

Context

IOC$ALLOC_CRAB conforms to the OpenVMS Alpha calling standard. Because IOC$ALLOC_CRAB calls EXE$ALONONPAGED to allocate sufficient memory for a CRAB, its caller cannot be executing above IPL$_POOL.

Arguments

item_cnt

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Number of items associated with the resource.

req_alloc_gran

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Requested allocation granularity associated with the resource.

crab_ref

VMS Usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of a cell to which IOC$ALLOC_CRAB returns the address of the allocated CRAB.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_BADPARAM	Specified allocation granularity is larger than the specified item count.
SS$_NORMAL	The routine completed successfully.
SS$_INSFMEM	Memory allocation request failed.

Description

A driver calls IOC$ALLOC_CRAB to allocate a counted resource allocation block (CRAB) that describes a counted resource. A counted resources, such as a set of map registers, has the following attributes:

- 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.

IOC$ALLOC_CRAB computes the size of the CRAB as the sum of the fixed portion of the CRAB, plus the maximum number of descriptors required in the allocation array. It then calls EXE$ALONONPAGED to allocate the CRAB. If the allocation request succeeds, IOC$ALLOC_CRAB initializes the CRAB as follows and returns SS$_NORMAL to its caller:

Field	Description
CRAB$W_SIZE	Size of the CRAB in bytes
CRAB$B_TYPE	DYN$C_MISC
CRAB$B_SUBTYPE	DYN$C_CRAB
CRAB$L_WQFL	CRAB$L_WQFL
CRAB$L_WQBL	CRAB$L_WQFL
CRAB$L_TOTAL_ITEMS	Contents of the **item_cnt** argument
CRAB$L_ALLOC_GRAN_MASK	One less than the contents of the **req_alloc_gran** argument (rounded up to the next highest power of two if the value specified is not a power of two)
CRAB$L_VALID_DESC_CNT	1
CRAB$L_SPINLOCK	Address of dynamic spin lock used to synchronize access to this CRAB. Currently, CRAB spin locks are obtained at IPL$_IOLOCK8.

IOC$ALLOC_CRAB initializes the first descriptor in the allocation array to indicate a set of **item_cnt** items of the resource, starting at item 0.

IOC$ALLOC_CRCTX

Allocates and initializes a counted resource context block (CRCTX).

Module

ALLOC_CNT_RES

Format

IOC$ALLOC_CRCTX crab ,crctx_ref ,fleck_index

Context

IOC$ALLOC_CRCTX conforms to the OpenVMS Alpha calling standard. Because IOC$ALLOC_CRCTX calls EXE$ALONONPAGED to allocate sufficient memory for a CRCTX, its caller cannot be executing above IPL$_POOL.

Arguments

crab

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of CRAB that describes the counted resource. For adapters that supply a counted resource, such as map registers, ADP$L_CRAB often contains this address.

crctx_ref

VMS Usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of a location in which IOC$ALLOC_CRCTX places the address of the allocated CRCTX.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_INSFMEM	Memory allocation request failed.

Description

A driver calls IOC$ALLOC_CRCTX to allocate a CRCTX to describe a specific request for a given counted resource, such as a set of map registers. The driver subsequently uses the CRCTX as input to IOC$ALLOC_CNT_RES to allocate a given set of the objects managed as a counted resource.

IOC$ALLOC_CRCTX calls EXE$ALONONPAGED to allocate the CRCTX. If the allocation request succeeds, IOC$ALLOC_CRCTX initializes the CRCTX as follows and returns SS$_NORMAL to its caller:

Field	Description
CRCTX$W_SIZE	Size of the CRCTX in bytes
CRCTX$B_TYPE	DYN$C_MISC
CRCTX$B_SUBTYPE	DYN$C_CRCTX
CRCTX$L_CRAB	Address of CRAB as specified in the **crab** argument
CRCTX$W_FSIZE	FKB$K_LENGTH
CRCTX$B_FTYPE	DYN$C_FRK
CRCTX$B_FLCK	IPL$_IOLOCK8

IOC$ALLOCATE_CRAM

Allocates a controller register access mailbox.

Module

CRAM-ALLOC

Macro

DPTAB (**ucb_crams** and **idb_crams** arguments) CRAM_ALLOC

Format

IOC$ALLOCATE_CRAM cram [,idb] [,ucb] [,adp]

Context

IOC$ALLOCATE_CRAM conforms to the OpenVMS Alpha calling standard. Because IOC$ALLOCATE_CRAM may need to allocate pages from the free page list, its caller must be executing at or below IPL$_SYNCH and must not hold spin locks ranked higher than IO_MISC.

IOC$ALLOCATE_CRAM acquires and releases the IO_MISC spin lock and returns to its caller at its caller's IPL.

Arguments

cram

VMS Usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of CRAM allocated by IOC$ALLOCATE_CRAM

idb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of IDB for device.

ucb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of UCB for device.

adp

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of ADP for device.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	CRAM has been successfully allocated.
SS$_INSFARG	Insufficient arguments supplied in call

Description

IOC$ALLOCATE_CRAM allocates a single controller register access mailbox (CRAM) and fills in the following fields:

CRAM$W_SIZE	Size of CRAM
CRAM$B_TYPE	Structure type (DYN$C_MISC)
CRAM$B_SUBTYPE	Structure type (DYN$C_CRAM)
CRAM$Q_RBADR	Address of remote tightly-coupled I/O interconnect (from IDB$Q_CSR)
CRAM$Q_HW_MBX	Physical address of hardware I/O mailbox
CRAM$L_MBPR	Mailbox pointer register (from ADP$PS_MBPR)
CRAM$Q_QUEUE_TIME	Default mailbox queue timeout value (from ADP$Q_QUEUE_TIME)
CRAM$Q_WAIT_TIME	Default mailbox wait-for-completion timeout value (from ADP$Q_WAIT_TIME)
CRAM$B_HOSE	Number of remote tightly-coupled I/O interconnect (from ADP$B_HOSE_NUM)
CRAM$L_IDB	IDB address
CRAM$L_UCB	UCB address

A driver may choose to allocate a CRAM on a per-controller or a per-unit basis. Typically a driver specifies values in the **idb_crams** and **ucb_crams** arguments of the DPTAB macro that indicate how many CRAMs should be allocated to a controller (IDB) or a unit (UCB). If these values (DPT$W_IDB_CRAMS and DPT$W_UCB_CRAMS) are nonzero in the DPT, the driver loading procedure automatically invokes IOC$ALLOCATE_CRAM to allocate the specified number of CRAMs. The driver-loading procedure thereafter sets up IDB$PS_CRAM to point to a linked list of CRAMs associated with a controller, UCB$PS_CRAM to a linked list of CRAMs associated with a device unit.

IOC$CANCEL_CNT_RES

Cancels a thread that has been stalled waiting for a counted resource.

Module

ALLOC_CNT_RES

Format

IOC$CANCEL_CNT_RES crab ,crctx [,resume_flag]

Context

IOC$CANCEL_CNT_RES conforms to the OpenVMS Alpha calling standard. Its caller must be executing at fork IPL, holding the corresponding fork lock.

Arguments

crab

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of CRAB that describes the counted resource. For adapters that supply a counted resource, such as map registers, ADP$L_CRAB often contains this address.

crctx

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of CRCTX structure that describes the request for the counted resource.

[resume_flag]

VMS Usage:	boolean
type:	longword (unsigned)
access:	read only
mechanism:	by value

Indication of whether the cancelled thread should be resumed. If true, IOC$CANCEL_CNT_RES calls the driver callback routine with SS$_CANCEL status. If not specified or false, IOC$CANCEL_CNT_RES does not resume the cancelled thread.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_BADPARAM	The specified CRCTX was not found in the CRAB wait queue.

Description

IOC$CANCEL_CNT_RES cancels a thread that has been stalled waiting for a counted resource. The resource request is described in the CRCTX structure; the counted resource itself is described in the CRAB.

IOC$CANCEL_CNT_RES scans the CRAB wait queue (CRAB$L_WFQL) to locate the specified CRCTX. If it cannot locate the CRCTX, it returns SS$_BADPARAM status to its caller.

If it locates the CRCTX in the CRAB wait queue and the **resume_flag** argument is not specified or is false, it removes the CRCTX from the queue and returns SS$_NORMAL status to its caller. Otherwise, after removing the CRCTX, it issues a `JSB` to the driver's callback routine (CRCTX$L_CALLBACK), passing it the following values:

Location	Contents
R0	SS$_CANCEL
R1	Address of CRAB
R2	Address of CRCTX
R3	CRCTX$Q_FR3
R4	CRCTX$Q_FR4
R5	CRCTX$Q_FR5
Other registers	Destroyed

The callback routine checks R0 to determine whether it has been called with SS$_NORMAL (from IOC$ALLOC_CNT_RES) or SS$_CANCEL status. If the latter, it takes appropriate steps to respond to the request cancellation. It must preserve all registers it uses other than R0 through R5 and exit with an `RSB` instruction.

When it regains control from the driver callback routine, IOC$CANCEL_CNT_RES returns SS$_NORMAL status to its caller.

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.

Module

[CPU*xxxx*]IO_SUPPORT_*xxxx*†

Macro

CRAM_CMD

Format

IOC$CRAM_CMD cmd_index ,byte_offset ,adp_ptr [,cram_ptr] [,buffer_ptr]

Context

IOC$CRAM_CMD conforms to the OpenVMS Alpha calling standard. It acquires no spin locks and leaves IPL unchanged. After inserting the hardware I/O mailbox values into the CRAM or specified buffer, IOC$CRAM_CMD returns to its caller.

Arguments

cmd_index

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Command index. IOC$CRAM_CMD uses this index to generate a mailbox command that is specific to the tightly-coupled interconnect that is to be the target of a request using this CRAM.

You can specify any of the following values (defined by the $CRAMDEF macro), although which of these I/O operations is supported depends on the I/O interconnect that is to be the object of the mailbox operation.

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

---

† where *xxxx* represents the internal OpenVMS code number for an Alpha CPU

Command Index	Description
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

byte_offset

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Byte offset of the field to be written or read from the base of device interface register (CSR) space. Calculation of the RBADR and MASK fields of the hardware mailbox depends on the addressing and masking mechanisms provided by the remote bus. The **byte_offset** argument is used by IOC$CRAM_CMD to calculate the RBADR.

adp_ptr

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of ADP associated with this command. IOC$CRAM_CMD uses this parameter to determine which tightly-coupled I/O interconnect is the object of the mailbox transaction and to construct the mailbox command accordingly.

cram_ptr

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of CRAM. IOC$CRAM_CMD returns the command, mask, and remote bus address values in the corresponding fields of the hardware I/O mailbox.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The calculated command, mask, and remote bus address values have been written to the CRAM and/or the specified buffer.
SS$_BADPARAM	Illegal command supplied as input or illegal argument supplied in call
SS$_INSFARG	Insufficient arguments supplied in call

Description

IOC$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. It performs the following tasks:

- Obtains the address of the command table specific to the given I/O interconnect from ADP$PS_COMMAND_TBL.
- Uses the value specified in the **command** argument as an index into the command table to determine the corresponding command supported by the I/O interconnect.
- If the command is valid for the I/O interconnect, IOC$CRAM_CMD writes it to CRAM$L_COMMAND, to the specified buffer, or to both. If the command is invalid for the I/O interconnect, IOC$CRAM_CMD returns SS$_BADPARAM status to its caller.
- Calculates the RBADR and MASK fields based of the hardware I/O mailbox, basing their values on the command, the address of device register interface space (ADP$Q_CSR or IDB$Q_CSR, if the **cram** argument is specified), the **byte_offset** argument, and interconnect-specific requirements. It writes these values to CRAM$B_BYTE_MASK and CRAM$Q_RBADR, to the specified buffer, or to both.
- Returns SS$_NORMAL status to its caller.

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.

Module

[SYSLOA]CRAM-IO

Macro

CRAM_IO

Format

IOC$CRAM_IO cram

Context

IOC$CRAM_IO conforms to the OpenVMS Alpha calling standard. It acquires no spin locks and leaves IPL unchanged. After queuing the request and waiting for its completion, IOC$CRAM_IO returns to its caller.

Arguments

cram

VMS Usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of CRAM associated with the hardware I/O mailbox transaction.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	CRAM has been successfully queued to the MBPR.
SS$_BADPARAM	Supplied argument is not a CRAM.
SS$_CTRLERR	Error bit set in mailbox transaction.
SS$_INSFARG	No argument supplied in call.
SS$_INTERLOCK	Failed to queue hardware I/O mailbox to MBPR in queue time.

SS$_TIMEOUT	Mailbox operation did not complete in mailbox transaction timeout interval.

Description

IOC$CRAM_IO performs an entire hardware I/O mailbox transaction from the queuing of the hardware I/O mailbox to the MBPR to the transaction's completion. A call to IOC$CRAM_IO is the equivalent of independent calls to IOC$CRAM_QUEUE and IOC$CRAM_WAIT. Prior to calling IOC$CRAM_IO, a driver typically calls IOC$CRAM_CMD to insert a command, mask, and remote interconnect address into the hardware I/O mailbox portion of the CRAM. For CRAMs involved in writes to device interface registers, the driver must also insert the data to be written into CRAM$Q_WDATA,

IOC$CRAM_IO initiates an I/O operation to a device in remote I/O space by writing the physical address of the hardware I/O mailbox portion of a CRAM to the MBPR. If it is not able to post the mailbox to the MBPR in the MBPR queue timeout interval (CRAM$Q_QUEUE_TIME), it returns SS$_INTERLOCK status to its caller.

If it does successfully queue the mailbox, it sets the CRAM$V_IN_USE bit in CRAM$B_CRAM_FLAGS and repeatedly checks the done bit in the hardware I/O mailbox (CRAM$V_MBX_DONE in CRAM$W_MBX_FLAGS):

- If the done bit is not set in the mailbox transaction timeout interval (CRAM$Q_WAIT_TIME), IOC$CRAM_IO leaves the CRAM$V_IN_USE bit in CRAM$B_CRAM_FLAGS set and returns SS$_TIMEOUT status to its caller.
- If the done bit is set, but the error bit in the mailbox (CRAM$V_MBX_ERROR in CRAM$W_MBX_FLAGS) is also set, IOC$CRAM_IO clears CRAM$V_IN_USE and returns SS$_CTRLERR status to its caller. Note that, if the disable-error bit (CRAM$V_DER) is set, IOC$CRAM_IO never returns an error (although it may request an IOMBXERR fatal bugcheck in the event of an error).
- If the done bit is set and the error bit is clear, IOC$CRAM_IO clears CRAM$V_IN_USE and returns SS$_NORMAL status to its caller. If IOC$CRAM_IO returns SS$_NORMAL status for read mailbox operations, the requested data has been returned to CRAM$Q_RDATA. A return of SS$_NORMAL status for mailbox write operations does not necessarily guarantee that the data placed in CRAM$Q_WDATA has been successfully written to the device register.

IOC$CRAM_QUEUE

Queues the hardware I/O mailbox defined within a controller register access mailbox (CRAM) to the mailbox pointer register (MBPR).

Module

[SYSLOA]CRAM-IO

Macro

CRAM_QUEUE

Format

IOC$CRAM_QUEUE cram

Context

IOC$CRAM_QUEUE conforms to the OpenVMS Alpha calling standard. It acquires no spin locks and leaves IPL unchanged. After queuing the request, IOC$CRAM_QUEUE returns to its caller. It is expected that the caller will eventually call IOC$CRAM_WAIT to await completion of the request.

Arguments

cram

VMS Usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of CRAM to be queued.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	CRAM has been successfully queued to the MBPR.
SS$_BADPARAM	Supplied argument is not a CRAM.
SS$_INSFARG	No argument supplied in call
SS$_INTERLOCK	Failed to queue hardware I/O mailbox to MBPR in queue time.

Description

IOC$CRAM_QUEUE initiates an I/O operation to a device in remote I/O space by writing the physical address of the hardware I/O mailbox portion of a CRAM to the MBPR. Prior to calling IOC$CRAM_QUEUE, a driver typically calls IOC$CRAM_CMD to insert a command, mask, and remote interconnect address into the hardware I/O mailbox portion of the CRAM. For CRAMs involved in writes to device interface registers, the driver must also insert the data to be written into CRAM$Q_WDATA,

If it is not able to post the mailbox to the MBPR in the MBPR queue timeout interval (CRAM$Q_QUEUE_TIME), IOC$CRAM_QUEUE returns SS$_INTERLOCK status to its caller. If the disable-error bit (CRAM$V_DER) is set, IOC$CRAM_QUEUE does not return an error (although it may request an IOMBXERR fatal bugcheck in the event of an error).

If IOC$CRAM_QUEUE does successfully queue the mailbox, it sets the CRAM$V_IN_USE bit in CRAM$B_CRAM_FLAGS and returns SS$_NORMAL.

IOC$CRAM_WAIT

Awaits the completion of a hardware I/O mailbox transaction to a tightly-coupled I/O interconnect.

Module

[SYSLOA]CRAM-IO

Macro

CRAM_WAIT

Format

IOC$CRAM_WAIT cram

Context

IOC$CRAM_WAIT conforms to the OpenVMS Alpha calling standard. It acquires no spin locks and leaves IPL unchanged. After queuing the request, IOC$CRAM_WAIT returns to its caller.

IOC$CRAM_WAIT assumes that its caller has previously called IOC$CRAM_QUEUE to post to the MBPR the hardware I/O mailbox defined within the specified CRAM for an I/O operation.

Arguments

cram

VMS Usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of CRAM associated with a previously-queued hardware I/O mailbox transaction.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	CRAM has been successfully queued to the MBPR.
SS$_BADPARAM	Supplied argument is not a CRAM.
SS$_CTRLERR	Error bit set in mailbox transaction.

SS$_INSFARG	No argument supplied in call.
SS$_TIMEOUT	Mailbox operation did not complete in mailbox transaction timeout interval.

Description

IOC$CRAM_WAIT checks the done bit in the hardware I/O mailbox (CRAM$V_MBX_DONE in CRAM$W_MBX_FLAGS):

- If CRAM$V_MBX_DONE is not set in the mailbox transaction timeout interval (CRAM$Q_WAIT_TIME), IOC$CRAM_WAIT leaves the CRAM$V_IN_USE bit in CRAM$B_CRAM_FLAGS set and returns SS$_TIMEOUT status to its caller.
- If CRAM$V_MBX_DONE is set, but the error bit in the mailbox (CRAM$V_MBX_ERROR in CRAM$W_MBX_FLAGS) is also set, IOC$CRAM_WAIT clears CRAM$V_IN_USE and returns SS$_CTRLERR status to its caller. In this case, CRAM$W_ERROR_BITS contains a device-specific encoding of additional status information.
- If the done bit is set and the error bit is clear, IOC$CRAM_WAIT clears CRAM$V_IN_USE and returns SS$_NORMAL status to its caller. If IOC$CRAM_WAIT returns SS$_NORMAL status for read mailbox operations, the requested data has been returned to CRAM$Q_RDATA. A return of SS$_NORMAL status for mailbox write operations does not necessarily guarantee that the data placed in CRAM$Q_WDATA has been successfully written to the device register.

Note

If the disable-error bit (CRAM$V_DER) is set, IOC$CRAM_WAIT does not return an error (although it may request an IOMBXERR fatal bugcheck in the event of an error).

IOC$DEALLOC_CNT_RES

Deallocates the requested number of items of a counted resource.

Module

DEALLOC_CNT_RES

Format

IOC$DEALLOC_CNT_RES crab ,crctx

Context

IOC$DEALLOC_CNT_RES conforms to the OpenVMS Alpha calling standard. Its caller must be executing at fork IPL, holding the corresponding fork lock.

Arguments

crab

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of CRAB.

crctx

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of CRCTX structure.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_BADPARAM	CRCTX$L_ITEM_CNT and CRCTX$L_ITEM_NUM fields are invalid.

Description

IOC$DEALLOC_CNT_RES deallocates a requested number of items of a counted resource. The resource request is described in the CRCTX structure; the counted resource itself is described in the CRAB. After deallocating the items, IOC$DEALLOC_CNT_RES attempts to restart any waiters for the resource.

IOC$DEALLOC_CNT_RES performs the following tasks:

1. It examines CRCTX$V_ITEM_VALID in CRCTX$L_FLAGS. If it is clear, IOC$DEALLOC_CNT_RES returns SS$_BADPARAM status to its caller.
2. It acquires the spin lock indicated by CRAB$L_SPINLOCK, raising IPL to IPL$_IOLOCKLL in the process.
3. It scans the CRAB allocation array for a descriptor into which the items being deallocated (indicated by CRCTX$L_ITEM_CNT) can be merged.
4. It adjusts the CRAB allocation array and CRAB$L_VALID_DESC_CNT to reflect the deallocation.
5. If there are waiters for the counted resource, IOC$DEALLOC_CNT_RES removes the CRCTX of the first waiter from the CRAB wait queue (CRAB$L_WQFL) and calls IOC$ALLOC_CNT_RES to grant the requested number of resources.

   If this attempt succeeds, IOC$DEALLOC_CNT_RES restores the context of the stalled waiter (R3 through R5), releases the spin lock indicated by CRAB$L_SPINLOCK (upon the condition that the caller of IOC$DEALLOC_CNT_RES did not already own this spin lock at the time of the call), and issues a standard call to the callback routine indicated by CRCTX$L_CALLBACK, passing it the address of the CRAB; the address of the CRCTX; the values stored in CRCTX$Q_FR3, CRCTX$Q_FR4, and CRCTX$Q_FR5; and SS$_NORMAL status.

   IOC$DEALLOC_CNT_RES continues to attempt to restart waiters in this manner until an allocation request fails. When this occurs, IOC$DEALLOC_CNT_RES replaces its CRCTX in the CRAB wait queue, conditionally releases the spin lock indicated by CRAB$L_SPINLOCK, and returns SS$_NORMAL status to its caller.
6. If there are no waiters for the counted resource, IOC$DEALLOC_CNT_RES conditionally releases the spin lock indicated by CRAB$L_SPINLOCK, and returns SS$_NORMAL status to its caller.

IOC$DEALLOC_CRAB

Deallocates a counted resource allocation block (CRAB).

Module

ALLOC_CNT_RES

Format

IOC$DEALLOC_CRAB crab

Context

IOC$DEALLOC_CRAB conforms to the OpenVMS Alpha calling standard. Because IOC$DEALLOC_CRAB calls EXE$DEANONPAGED, its caller cannot be executing above IPL$_SYNCH.

Arguments

crab

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of CRAB to be deallocated.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.

Description

A driver calls IOC$DEALLOC_CRAB to deallocate a CRAB. IOC$DEALLOC_CRAB passes the address of the CRAB to EXE$DEANONPAGED and returns SS$_NORMAL status to its caller.

IOC$DEALLOC_CRCTX

Deallocates a counted resource context block (CRCTX).

Module

ALLOC_CNT_RES

Format

IOC$DEALLOC_CRCTX crctx

Context

IOC$DEALLOC_CRCTX conforms to the OpenVMS Alpha calling standard. Because IOC$DEALLOC_CRCTX calls EXE$DEANONPAGED, its caller cannot be executing above IPL$_SYNCH.

Arguments

crctx

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of CRCTX to be deallocated.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.

Description

A driver calls IOC$DEALLOC_CRCTX to deallocate a CRCTX. IOC$DEALLOC_CRCTX passes the address of the CRCTX to EXE$DEANONPAGED and returns SS$_NORMAL status to its caller.

IOC$DEALLOCATE_CRAM

Deallocates a controller register access mailbox.

Module

CRAM-ALLOC

Macro

CRAM_DEALLOC

Format

IOC$DEALLOCATE_CRAM cram

Context

IOC$DEALLOCATE_CRAM conforms to the OpenVMS Alpha calling standard. Its caller must be executing at or below IPL 8 and must not hold spin locks ranked higher than IO_MISC.

IOC$DEALLOCATE_CRAM acquires and releases the IO_MISC spin lock and returns to its caller at its caller's IPL.

Arguments

cram

VMS Usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of CRAM to be deallocated by IOC$DEALLOCATE_CRAM

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	CRAM has been successfully deallocated.
SS$_BADPARAM	Supplied argument is not a CRAM.
SS$_INSFARG	Insufficient arguments supplied in call

Description

IOC$DEALLOCATE_CRAM deallocates a single controller register access mailbox (CRAM).

IOC$KP_REQCHAN

Stalls a kernel process in such a manner that it can be resumed by the granting of a device controller channel.

Module

KERNEL_PROCESS_MIN, KERNEL_PROCESS_MON

Macro

KP_STALL_REQCHAN

Format

IOC$KP_REQCHAN kpb ,priority

Context

IOC$KP_REQCHAN conforms to the OpenVMS Alpha calling standard. It can only be called by a kernel process.

A kernel process calls IOC$KP_REQCHAN at fork IPL holding the appropriate fork lock.

If the requested channel is busy, either the channel-requesting routine IOC$PRIMITIVE_REQCHANH or IOC$PRIMITIVE_REQCHANL preserves the contents of its caller's R3 in UCB$Q_FR3 (contents of caller's R3). IOC$RELCHAN eventually issues a JSB instruction to the fork routine upon granting the channel request. At this time, the kernel process is provided with the contents of UCB$Q_FR3 in R3, the IDB address in R4, and the UCB address in R5.

Arguments

kpb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of the caller's KPB which must be a VEST KPB. KPB$PS_UCB must contain the address of a UCB and KPB$PS_IRP must contain the address of an IRP.

priority

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Priority of the request for the controller channel. You must specify one of the following symbolic constants:

Constant	Meaning
KPB$K_LOW	Insert fork block of UCB requesting controller channel at the tail of the channel-wait queue.
KPB$K_HIGH	Insert fork block of UCB requesting controller channel at the head of the channel-wait queue.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_BADPARAM	The **kpb** argument does not specify a VEST KPB, or an illegal value was supplied in the **priority** argument.
SS$_INSFARG	Not all of the required arguments were specified.

Description

IOC$KP_REQCHAN first checks the CRB to determine if the controller channel is busy. If the CRB is not busy (CRB$V_BSY in CRB$B_MASK is clear), IOC$KP_REQCHAN grants the channel request immediately by placing the UCB address in IDB$L_OWNER and returning SS$_NORMAL status to its caller.

If the CRB is busy, IOC$KP_REQCHAN performs the following tasks to initiate a stall of the kernel process:

1. Copies the **priority** argument to KPB$IS_CHANNEL_DATA.
2. Inserts the procedure descriptor of subroutine STALL_REQCHAN in KPB$PS_SCH_STALL_RTN, thus making it the kernel process scheduling stall routine.
3. Clears KPB$PS_SCH_RESTART, thus indicating that there is no kernel process scheduling restart routine.
4. Calls EXE$KP_STALL_GENERAL, passing to it the address of the KPB.

Note that, having stalled the kernel process, the STALL_REQCHAN kernel process scheduling stall routine returns control to EXE$KP_STALL_GENERAL, which returns to the initiator of the kernel process thread (that is, the caller of EXE$KP_START or EXE$KP_RESTART). When the controller channel request is ultimately granted, STALL_REQCHAN calls EXE$KP_RESTART which, in turn, passes control back to IOC$KP_REQCHAN. IOC$KP_REQCHAN then returns to the kernel process that called it.

IOC$KP_WFIKPCH, IOC$KP_WFIRLCH

Stall a kernel process in such a manner that it can be resumed by device interrupt processing.

Module

KERNEL_PROCESS_MIN, KERNEL_PROCESS_MON

Macro

KP_STALL_WFIKPCH
KP_STALL_WFIRLCH

Format

IOC$KP_WFIKPCH kpb ,time ,newipl

IOC$KP_WFIRLCH kpb ,time ,newipl

Context

IOC$KP_WFIKPCH and IOC$KP_WFIRLCH conform to the OpenVMS Alpha calling standard. They can only be called by a kernel process.

When called, IOC$KP_WFIKPCH or IOC$KP_WFIRLCH assumes that the local processor has obtained the appropriate synchronization with the device database by securing the appropriate device lock, as recorded in the unit control block (UCB$L_DLCK) of the device unit from which the interrupt is expected. This requirement also presumes that the local processor is executing at the device IPL associated with the lock.

Before exiting, the wait-for-interrupt routine (IOC$PRIMITIVE_WFIKPCH or IOC$PRIMITIVE_WFIRLCH) conditionally releases the device lock, so that if the initiator of the kernel process thread previously owned the device lock, it will continue to hold it when it regains control. IOC$PRIMITIVE_WFIKPCH or IOC$PRIMITIVE_WFIRLCH also lowers the local processor's IPL to the IPL specified in the **newipl** argument.

Arguments

kpb

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of the caller's KPB (which must be a VEST KPB). KPB$PS_UCB must contain the address of a UCB and KPB$PS_IRP must contain the address of an IRP.

time

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Timeout value in seconds.

newipl

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

IPL to which to lower before returning to the initiator of the kernel process thread (that is, the caller of EXE$KP_START or EXE$KP_RESTART). This IPL must be the fork IPL associated with device processing and at which the kernel process was executing prior to invoking the DEVICELOCK macro.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_BADPARAM	The **kpb** argument does not specify a VEST KPB.
SS$_INSFARG	Not all of the required arguments were specified.
SS$_TIMEOUT	A timeout has occurred.

Description

IOC$KP_WFIKPCH and IOC$KP_WFIRLCH perform the following tasks to initiate a stall of the kernel process:

1. Copy the **time** argument to KPB$IS_TIMEOUT_TIME and the **newipl** argument to KPB$IS_RESTORE_IPL.
2. Move the symbolic constant KPB$K_KEEP (for IOC$KP_WFIKPCH) or KPB$K_RELEASE (for IOC$KP_WFIRLCH) to KPB$IS_CHANNEL_DATA.
3. Insert the procedure descriptor of subroutine STALL_WFIXXCH in KPB$PS_SCH_STALL_RTN, this making it the kernel process scheduling stall routine.
4. Clear KPB$PS_SCH_RESTART, thus indicating that there is no kernel process scheduling restart routine.
5. Call EXE$KP_STALL_GENERAL, passing to it the address of the KPB.

Note that, having stalled the kernel process, the STALL_WFIXXCH kernel process scheduling stall routine returns control to EXE$KP_STALL_GENERAL, which returns to the initiator of the kernel process thread (that is, the caller of EXE$KP_START or EXE$KP_RESTART). When interrupt servicing transfers control back to STALL_WFIXXCH, or a timeout occurs, STALL_WFIXXCH calls EXE$KP_RESTART which, in turn, passes control back to IOC$KP_WFIKPCH or IOC$KP_WFIRLCH. The kernel process wait-for-interrupt stall routine then returns to the kernel process that called it.

IOC$LOAD_MAP

Loads a set of adapter-specific map registers.

Module

[CPU*xxxx*]MAPREG_*xxxx*†

Format

IOC$LOAD_MAP adp ,crctx ,svapte ,boff ,dma_address_ref

Context

IOC$LOAD_MAP conforms to the OpenVMS Alpha calling standard.

Arguments

adp

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of ADP for adapter which provides the map registers.

crctx

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of CRCTX that describes a map register allocation (that is, a CRCTX that has been obtained by a call to IOC$ALLOC_CRCTX and supplied in a call to IOC$ALLOC_CNT_RES for the CRAB that manages this adapter's map registers).

svapte

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

System virtual address of the PTE for the first page to be used in the transfer.

boff

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Byte offset into the first page of the transfer buffer.

---

† where *xxxx* represents the internal OpenVMS code number for an Alpha CPU

dma_address_ref

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of a location to receive a port-specific DMA address. For DEC 3000-500 systems, this address is a function of the starting map register and the byte offset. A DEC 3000-500 system port driver must strip off two lower bits when loading the address register of the DMA device.

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_INSMEM	Memory allocation failure.

Description

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

IOC$LOAD_MAP computes a port-specific DMA address and returns it to the driver for use in a hardware I/O mailbox operation that loads the address register of a DMA device.

IOC$MAP_IO

IOC$MAP_IO 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 swizzle space, dense space, or sparse space.

IOC$MAP_IO is supported on PCI, EISA, TURBOchannel, or PCI systems. It is not supported on XMI systems.

Description

The routine prototype is as follows:

```
int ioc$map_io (ADP     *adp,
                int     node,
                uint64  *physical_offset,
                int     num_bytes,
                int     attributes,
                uint64  *iohandle)
```

```
Inputs

    adp     Address of bus ADP.  Driver can get this from
            IDB$PS_ADP.

    node    Bus node number of device.  Bus specific
            interpretation.  Available to driver in
            CRB$L_NODE (driver must be loaded with /NODE
            qualifier).

    physical_offset Address of a quadword cell.  For EISA, PCI,
            and Futurebus, the quadword cell should contain
            the starting bus physical address to be mapped.  For
            Turbochannel, the quadword cell should contain the
            physical offset from the Turbochannel slot base
            address.

    num_bytes   Number of bytes to be mapped.  Expressed in terms
            of the bus/device without regard to the platform
            hardware addressing tricks.

    attributes  Specifies desired attributes of space to be
            mapped.  From [lib]iocdef.  One of the following:

            IOC$K_BUS_IO_BYTE_GRAN

                Request mapping in a platform address space which
                corresponds to bus I/O space and provides byte
                granularity access.  In general, if you are mapping
                device control registers that exist in bus I/O space,
                you should specify this attribute.  For example,
                drivers for PCI devices with registers in PCI I/O
                space or EISA devices with EISA I/O port addresses
                should request mapping with this attribute.

            IOC$K_BUS_MEM_BYTE_GRAN

                Request mapping in a platform address space which
                corresponds to bus memory space and provides byte
                granularity access.  In general, if you are mapping
                device registers that exist in bus memory space, you
                should specify this attribute.  For example, drivers
                for PCI devices with registers in PCI memory space
                should request mapping with this attribute.

            IOC$K_BUS_DENSE_SPACE
```

Request mapping in a platform address space that corresponds to bus memory space and provides coarse access granularity. IOC$K_BUS_DENSE_SPACE is suitable for mapping device memory buffers such as graphics frame buffers. In IOC$K_BUS_DENSE_SPACE, there must be no side effects on reads and it may be possible for the processor to merge writes. Thus you should not map device registers in dense space.

iohandle	Pointer to a 64 bit cell. A 64 bit magic number is written to this cell by IOC$MAP_IO when the mapping request is successful. The caller must save the iohandle, as it is an input to IOC$CRAM_CMD and to the new platform independent access routines IOC$READ_IO and IOC$WRITE_IO.

Outputs

SS$_NORMAL	Success. The address space is mapped. A 64 bit IOHANDLE is written to the caller’s buffer.
SS$_BADPARAM	Bad input argument. For example, the requested bus address may not be accessible from the CPU, or the attribute may be unrecognized.
SS$_UNSUPPORTED	Address space with the requested attributes not available on this platform. For example, the Jensen platform does not support EISA memory dense space.
SS$_INSFSPTS	Not enough PTEs to satisfy mapping request.

IOC$NODE_FUNCTION

Performs node-specific functions on behalf of a driver, such as enabling or disabling interrupts from a bus slot.

Module

[SYSLOA]MISC_SUPPORT

Format

IOC$NODE_FUNCTION crb_addr ,function_code

Context

IOC$NODE_FUNCTION conforms to the OpenVMS Alpha calling standard. It may be called in kernel mode at any IPL and may acquire the MEGA spin lock (SPL$C_MEGA), raising IPL to IPL$_MEGA in the process, depending on the function code.

Arguments

crb_addr

VMS Usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of CRB.

function_code

VMS Usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Function to be effected for the bus node indicated by the **crb_addr** argument. You can specify one of the following values (defined by the $IOCDEF macro in SYS$LIBRARY:LIB.MLB). Note that not all function codes are supported by all adapters.

Code	Action
IOC$K_ENABLE_INTR	Enable interrupts
IOC$K_DISABLE_INTR	Disable interrupts
IOC$K_ENABLE_SG	Enable scatter/gather map
IOC$K_DISABLE_SG	Disable scatter/gather map
IOC$K_ENABLE_PAR	Enable parity
IOC$K_DISABLE_PAR	Disable parity
IOC$K_ENABLE_BLKM	Enable block mode
IOC$K_DISABLE_BLKM	Disable block mode

Returns

VMS Usage:	cond_value
type:	longword_unsigned
access:	longword (unsigned)
mechanism:	write only—by value

Status indicating the success or failure of the operation.

Return Values

SS$_NORMAL	The routine completed successfully.
SS$_ILLIOFUNC	Requested function not available on this platform or bus.

Description

IOC$NODE_FUNCTION locates the ADP associated with the specified CRB (from VEC$PS_ADP) and calls the adapter-specific node function routine specified in ADP$PS_NODE_FUNCTION. The node function routine performs the function indicated by the **function_code** argument.

Drivers request the node-specific functions as follows:

- IOCK_ENABLE_INTR, IOCK_DISABLE_INTR

  On both DEC 3000-500 and DEC 3000-300 systems, when the console transfers control to OpenVMS Alpha, TURBOchannel interrupts from all slots are disabled. The controller or unit initialization routine of a driver for a TURBOchannel devices must call IOC$NODE_FUNCTION, specifying the IOC$K_ENABLE_INTR function code, to enable interrupts for the TURBOchannel slot in which the device resides. The field CRB$L_NODE of the specified CRB contains this slot number.

  Calling IOC$NODE_FUNCTION with the IOC$K_DISABLE_INTR code disables interrupts from the node.

- IOCK_ENABLE_SG, IOCK_DISABLE_SG

  On DEC 3000-500 systems, calling IOC$NODE_FUNCTION with function code IOC$K_ENABLE_SG, allows DMA transactions from a device to use the DEC 3000-500 system scatter/gather map. The TURBOchannel slot of the device is indicated by the field CRB$L_NODE in the specified CRB.

  Calling IOC$NODE_FUNCTION with the IOC$K_DISABLE_SG code disables the scatter/gather map.

  DEC 3000-300 systems have no scatter/gather map. IOC$NODE_FUNCTION returns SS$_ILLIOFUNC if it is called on a DEC 3000-300 system with either an IOC$K_ENABLE_SG or IOC$K_DISABLE_SG function code.

- IOCK_ENABLE_PAR, IOCK_DISABLE_PAR

  On DEC 3000-500 systems, calling IOC$NODE_FUNCTION with function code IOC$K_ENABLE_PAR causes parity to be generated on TURBOchannel transactions directed to a device, and causes parity to be checked on TURBOchannel transactions coming from the device. The TURBOchannel slot of the device is indicated by the field CRB$L_NODE in the specified CRB.

If an adapter supports TURBOchannel parity, a driver controller or unit initialization routine enable it by calling IOC$NODE_FUNCTION with the IOC$K_ENABLE_PAR function code.

Calling IOC$NODE_FUNCTION with the IOC$K_DISABLE_PAR code disables TURBOchannel parity.

DEC 3000-300 systems do not support TURBOchannel parity. IOC$NODE_FUNCTION returns SS$_ILLIOFUNC if it is called on a DEC 3000-300 system with either an IOC$K_ENABLE_PAR or IOC$K_DISABLE_PAR function code.

- IOCK_ENABLE_BLKM, IOCK_DISABLE_BLKM

  On DEC 3000-500 systems, calling IOC$NODE_FUNCTION with function code IOC$K_ENABLE_BLKM causes block mode to be used on TURBOchannel transactions to and from the device indicated by the field CRB$L_NODE in the specified CRB. Most drivers have no need to enable block mode.

  DEC 3000-300 systems do not support TURBOchannel block mode. IOC$NODE_FUNCTION returns SS$_ILLIOFUNC if it is called on a DEC 3000-300 system with either an IOC$K_ENABLE_BLKM or IOC$K_DISABLE_BLKM function code.

IOC$READ_IO

Reads a value from a previously mapped location in I/O address space. This routine requires that the I/O space to be accessed has been previously mapped by a call to IOC$MAP_IO.

IOC$READ_IO is supported on PCI, EISA, TURBOchannel, and PCI systems. It is not supported on XMI systems.

Description

The routine prototype for IOC$READ_IO is as follows:

```
int ioc$read_io (ADP    *adp,
                 uint64 *iohandle,
                 int    offset,
                 int    length,
                 void   *read_data)

Inputs

    adp     Address of bus ADP.  Driver can get this from
            IDB$PS_ADP.

    iohandle Pointer to a 64 bit IOHANDLE.  The 64 bit IOHANDLE
            is obtained by calling the platform independent
            mapping routine IOC$MAP_IO.

    offset  Offset in device space of field to be read or written.
            This should be specified as an offset from the base of
            the space that was previously mapped by the call to
            IOC$MAP_IO.  The offset is specified in terms of the
            device or bus without regard to any hardware address
            trickery.

    length  Length of field to be read or written.  Should be 1
            (byte), 2 (word), 3 (tribyte), 4 (longword) or 8
            (quadword).  Note that not all of these lengths are
            supported on all buses.

    read_data Pointer to a data cell.  For ioc$read_io, the
            data read from the device will be returned in this cell.
            If the requested data length was 1, 2, 3, or 4, a
            longword is written to the data cell with valid data
            in the byte lane(s) corresponding to the requested
            length and offset.  If the requested data length was 8,
            a quadword is written to the data cell.

    write_data Pointer to a data cell.  The data cell should
            contain the data to be written to the device.  For
            lengths of 1, 2, 3 or 4, the ioc$write_io routine
            reads a longword from the data cell and writes this
            longword to the bus with the proper byte enables set
            according to the length and offset.  The actual
            data to be written must be positioned in the proper
            byte lane(s) according to the requested length and
            offset.  For a length 8 transfer, the ioc$write_io
            routine reads a quadword from the data cell.

Outputs

    SS$_NORMAL Success.  If IOC$READ_IO, data is returned in the
            caller's buffer.  If IOC$WRITE_IO, data is written to
            device.

    SS$_BADPARAM Bad input argument, such as an illegal length.
```

SS$_UNSUPPORTED	A transaction length not supported by this bus or platform.

IOC$UNMAP_IO

Unmaps a previously mapped I/O address space, returning the IOHANDLE and the PTEs to the system. The caller's quadword cell containing the IOHANDLE is cleared.

Description

The routine prototype is as follows:

```
int ioc$unmap_io (ADP   *adp,
                  uint64 *iohandle)
```

IOC$WRITE_IO

Writes a value to a previously mapped location in I/O address space. IOC$WRITE_IO requires that the I/O space to be accessed has been previously mapped by a call to IOC$MAP_IO.

Description

The routine prototype is as follows:

```
int ioc$write_io (ADP    *adp,
                  uint64 *iohandle,
                  int    offset,
                  int    length,
                  void   *write_data)

Inputs

    adp     Address of bus ADP.  Driver can get this from
            IDB$PS_ADP.

    iohandle Pointer to a 64 bit IOHANDLE.  The 64 bit IOHANDLE
            is obtained by calling the platform independent
            mapping routine IOC$MAP_IO.

    offset  Offset in device space of field to be read or written.
            This should be specified as an offset from the base of
            the space that was previously mapped by the call to
            IOC$MAP_IO.  The offset is specified in terms of the
            device or bus without regard to any hardware address
            trickery.

    length  Length of field to be read or written.  Should be 1
            (byte), 2 (word), 3 (tribyte), 4 (longword) or 8
            (quadword).  Note that not all of these lengths are
            supported on all buses.

    read_data Pointer to a data cell.  For ioc$read_io, the
            data read from the device will be returned in this cell.
            If the requested data length was 1, 2, 3, or 4, a
            longword is written to the data cell with valid data
            in the byte lane(s) corresponding to the requested
            length and offset.  If the requested data length was 8,
            a quadword is written to the data cell.

    write_data Pointer to a data cell.  The data cell should
            contain the data to be written to the device.  For
            lengths of 1, 2, 3 or 4, the ioc$write_io routine
            reads a longword from the data cell and writes this
            longword to the bus with the proper byte enables set
            according to the length and offset.  The actual
            data to be written must be positioned in the proper
            byte lane(s) according to the requested length and
            offset.  For a length 8 transfer, the ioc$write_io
            routine reads a quadword from the data cell.

Outputs

    SS$_NORMAL Success.  If ioc$read_io, data is returned in the
            caller’s buffer.  If ioc$write_io, data is written to
            device.

    SS$_BADPARAM Bad input argument, such as an illegal length.

    SS$_UNSUPPORTED A transaction length not supported by this bus
            or platform.
```

IOC_STD$ALTREQCOM

Completes an I/O request for a device using the disk or tape class drivers.

Module

IOSUBNPAG

Format

IOC_STD$ALTREQCOM (iost1, iost2, cdrp, irp_p, ucb_p)

Arguments

Argument	Type	Access	Mechanism	Status
iost1	integer	input	value	required
iost2	integer	input	value	required
cdrp	CDRP	input	reference	required
irp_p	pointer	output	reference	required
ucb_p	pointer	output	reference	required

iost1
First longword of I/O status.

iost2
Second longword of I/O status.

cdrp
Class driver request packet.

irp_p
Address at which IOC_STD$ALTREQCOM writes the address of the I/O request packet.

ucb_p
Address at which IOC_STD$ALTREQCOM writes the address of the unit control block.

Context

IOC_STD$ALTREQCOM is typically called at fork IPL with the corresponding fork lock held in an OpenVMS multiprocessing system.

Description

For Digital internal use only.

Macro

CALL_ALTREQCOM

In an Alpha driver, the CALL_ALTREQCOM macro calls IOC_STD$ALTREQCOM, using the current contents of R0, R1, and R5 as the **iost1**, **iost2**, and **cdrp** arguments, respectively. When IOC_STD$ALTREQCOM returns, the macro returns the address of the IRP in R3 and the address of the UCB in R4.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$ALTREQCOM replaces IOC$ALTREQCOM (used by OpenVMS VAX drivers). Unlike IOC$ALTREQCOM, IOC_STD$ALTREQCOM does not return the addresses of the IRP and UCB in R3 and R5, respectively.

IOC_STD$BROADCAST

Broadcasts the specified message to a given terminal.

Module

IOSUBNPAG

Format

status = IOC_STD$BROADCAST (msglen, msg_p, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
msglen	integer	input	value	required
msg_p	address	input	reference	required
ucb	UCB	input	reference	required

msglen
Message length.

msg_p
Message.

ucb
Address of target terminal's UCB.

Return Values

SS$_ILLIOFUNC	The specified **term_ucb** is not associated with a terminal.
SS$_INSFMEM	Insufficient dynamic nonpaged pool to satisfy the request.
SS$_NORMAL	The broadcast completed successfully.

Context

IOC_STD$BROADCAST is typically called at fork IPL with the corresponding fork lock held in an OpenVMS multiprocessing system.

Description

For Digital internal use only.

Macro

CALL_BROADCAST [save_r1]

where:

> **save_r1** indicates that the macro should preserve register R1 across the call to IOC_STD$BROADCAST. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

In an Alpha driver, the CALL_BROADCAST macro calls IOC_STD$BROADCAST, using the current contents of R1, R2, and R5 as the **msglen**, **msg_p**, and **ucb** arguments, respectively. It returns status in R0. Unless you specify **save_r1**=**NO**, the macro preserves the quadword register R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$BROADCAST replaces IOC$BROADCAST (used by OpenVMS VAX drivers). Unlike IOC$BROADCAST, IOC_STD$BROADCAST does not preserve R1 across the call.

IOC_STD$CANCELIO

Conditionally marks a UCB so that its current I/O request will be canceled.

Module

IOSUBNPAG

Format

IOC_STD$CANCELIO (chan, irp, pcb, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
chan	integer	input	value	required
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required

chan
Channel index number.

irp
I/O request packet. IOC_STD$CANCELIO reads the following IRP fields:

Field	Contents
IRP$L_PID	Process identification of the process that queued the I/O request
IRP$L_CHAN	I/O request channel index number

pcb
Current process control block.

ucb
Unit control block. IOC_STD$CANCELIO reads UCB$L_STS to determine if the device is busy (UCB$V_BSY set) or idle (UCB$V_BSY clear). IOC_STD$CANCELIO sets UCB$V_CANCEL if the I/O request should be canceled.

Context

IOC_STD$CANCELIO executes at its caller’s IPL, obtains no spin locks, and returns control to its caller at the caller’s IPL. It is usually called by EXE$CANCEL (if specified in the DDT as the driver’s cancel-I/O routine) at fork IPL, holding the corresponding fork lock in a multiprocessing environment.

Description

IOC_STD$CANCELIO cancels I/O to a device in the following device-independent manner:

1. It confirms that the device is busy by examining the device-busy bit in the UCB status longword (UCB$V_BSY in UCB$L_STS).
2. It confirms that the IRP in progress on the device originates from the current process (that is, the contents of IRP$L_PID and PCB$L_PID are identical).
3. It confirms that the specified channel-index number is the same as the value stored in the IRP’s channel-index field (IRP$L_CHAN).
4. It sets the cancel-I/O bit in the UCB status longword (UCB$V_CANCEL in UCB$L_STS).

Macro

```
CALL_CANCELIO [save_r0r1]
```

where:

> **save_r0r1** indicates that the macro should preserve registers R0 and R1 across the call to IOC_STD$CANCELIO. If **save_r0r1** is blank or **save_r0r1=YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1=NO**, the registers are not saved.)

In an Alpha driver, the CALL_CANCELIO macro calls IOC_STD$CANCELIO, using the current contents of R2, R3, R4, and R5 as the **chan**, **irp**, **pcb**, and **ucb** arguments, respectively. Unless you specify **save_r0r1=NO**, it preserves the quadword registers R0 and R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$CANCELIO replaces IOC$CANCELIO (used by OpenVMS VAX drivers). Unlike IOC$CANCELIO, IOC_STD$CANCELIO does not preserve R0 and R1 across the call.

IOC_STD$CLONE_UCB

Copies a template UCB and links it to the appropriate DDB list.

Module

UCBCREDEL

Format

status = IOC_STD$CLONE_UCB (tmpl_ucb, new_ucb_p)

Arguments

Argument	Type	Access	Mechanism	Status
tmpl_ucb	UCB	input	reference	required
new_ucb_p	pointer	output	reference	required

tmpl_ucb
Template unit control block.

new_ucb_p
Location into which IOC_STD$CLONE_UCB writes the address of the newly-created unit control block.

Return Values

SS$_NORMAL	UCB cloning was successful.
SS$_INSFMEM	Insufficient nonpaged pool to copy UCB.

Context

A driver calls IOC_STD$CLONE_UCB at or below IPL$_MAILBOX with the I/O database locked for write access.

Description

For Digital internal use only.

Macro

CALL_CLONE_UCB [interface_warning=YES]

where:

> **interface_warning**=**YES**, the default, specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the VAX version of the corresponding system routine. **interface_warning**=**NO** suppresses the warning.

In an OpenVMS Alpha driver, CALL_CLONE_UCB calls IOC_STD$CLONE_UCB using the current contents of R5 as the **tmpl_ucb** argument. CALL_CLONE_UCB returns status in R0 and the address of the newly-created UCB in R2, but does not return the address of the UCBs that precede and follow it on the DDB chain in R3 and R1, respectively.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$CLONE_UCB replaces IOC$CLONE_UCB (used by OpenVMS VAX drivers). IOC_STD$CLONE_UCB does not return the addresses of the UCBs that precede and follow the newly-created UCB on the DDB chain.

IOC_STD$COPY_UCB

Copies and initializes a template UCB and ORB.

Module

UCBCREDEL

Format

status = IOC_STD$COPY_UCB (src_ucb, new_ucb)

Arguments

Argument	Type	Access	Mechanism	Status
src_ucb	UCB	input	reference	required
new_ucb	pointer	output	reference	required

src_ucb
Template unit control block.

new_ucb
Location into which IOC_STD$COPY_UCB writes the address of the newly-created duplicate unit control block.

Return Values

SS$_NORMAL	UCB copy was successful.
SS$_INSFMEM	Insufficient nonpaged pool to copy UCB.

Context

A driver calls IOC_STD$COPY_UCB at or below IPL$_MAILBOX with the I/O database locked for write access.

Description

For Digital internal use only.

Macro

CALL_COPY_UCB

In an Alpha driver, CALL_COPY_UCB calls IOC_STD$COPY_UCB using the current contents of R5 as the **src_ucb** argument. CALL_CLONEUCB returns the address of the newly-created UCB in R2.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note that IOC_STD$COPY_UCB replaces IOC$COPY_UCB (used by OpenVMS VAX drivers). IOC_STD$COPY_UCB does not preserve the contents of R3 and R4 across the call.

IOC_STD$CREDIT_UCB

Credits the UCB charges associated with a given UCB against the process identified by the contents of UCB$L_CPID.

Module

UCBCREDEL

Format

IOC_STD$CREDIT_UCB (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block.

Context

A driver calls IOC_STD$CREDIT_UCB at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

CALL_CREDIT_UCB

In an Alpha driver, CALL_CREDIT_UCB calls IOC_STD$CREDIT_UCB using the current contents of R5 as the **ucb** argument.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$CREDIT_UCB replaces IOC$CREDIT_UCB (used by OpenVMS VAX drivers).

IOC_STD$CVT_DEVNAM

Converts a device name and unit number to a physical device name string.

Module

IOSUBNPAG

Format

status = IOC_STD$CVT_DEVNAM (buflen, buf, form, ucb, outlen_p)

Arguments

Argument	Type	Access	Mechanism	Status
buflen	integer	input	value	required
buf	address	input	reference	required
form	integer	input	value	required
ucb	UCB	reference	input	required
outlen_p	pointer	output	reference	required

buflen
Size of output buffer in bytes.

buf
Output buffer.

form
Name string formation mode, as follows:

Mode	Description
-2 (DVI$_DISPLAY_ DEVNAM)	Name suitable for displays but not suitable for $ASSIGN: "$alloclass$ddcn: (host1[, host2])", "node$ddcn", or "ddcn"
–1 (DVI$_DEVNAM)	Name suitable for displays: "node$ddcn" for non-local devices or "node$ddcn" or "ddcn" for local devices
0 (DVI$_ FULLDEVNAM)	Name with appropriate node information: either "$alloclass$ddcn" or "node$ddcn"
1 (DVI$_ ALLDEVNAM)	Name with allocation class information: either "$alloclass$ddcn" or "node$ddcn"
2 (no GETDVI item code)	Old-fashioned name: "ddcn"
3 (no GETDVI item code)	Secondary path name for displays (same as –1 except secondary path name is returned)

Mode	Description
4 (no GETDVI item code)	Path controller name for displays (same as –1 except no unit number is appended)

ucb
Unit control block for device.

outlen_p
Address of location in which IOC_STD$CVT_DEVNAM returns the length of the conversion string.

Return Values

SS$_BUFFEROVF	Successful completion, but specified buffer cannot hold the entire device name string.
SS$_NORMAL	Normal, successful completion.

Context

IOC_STD$CVT_DEVNAM is typically called at fork IPL with the corresponding fork lock held in an OpenVMS multiprocessing system.

Description

For Digital internal use only.

Macro

CALL_CVT_DEVNAM

In an Alpha driver, the CALL_CVT_DEVNAM macro calls IOC_STD$CVT_DEVNAM, using the current contents of R0, R1, R4, and R5 as the **buflen**, **buf**, **form**, and **ucb** arguments, respectively. When IOC_STD$CVT_DEVNAM returns, the macro returns status in R0 and the length of the conversion string in R1.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$CVT_DEVNAM replaces IOC$CVT_DEVNAM (used by OpenVMS VAX drivers). Unlike IOCCVT_DEVNAM, IOC_STDCVT_DEVNAM does not return the length of the conversion string in R1.

IOC_STD$CVTLOGPHY

Conditionally converts a logical block number to a physical disk address and stores the result in the I/O request packet.

Module

IOSUBRAMS

Format

IOC_STD$CVTLOGPHY (lbn, irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
lbn	integer	input	value	required
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

lbn
Logical block number to be converted.

irp
I/O request packet.

ucb
Unit control block.

Context

A driver calls IOC_STD$CVTLOGPHY at fork IPL with the corresponding fork lock held in a multiprocessing system.

Description

For Digital internal use only.

Macro

CALL_CVTLOGPHY

In an Alpha driver, the CALL_CVTLOGPHY macro calls IOC_STD$CVTLOGPHY, using the current contents of R0, R3, and R5 as the **lbn**, **irp** and **ucb** arguments, respectively.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$CVTLOGPHY replaces IOC$CVTLOGPHY (used by OpenVMS VAX drivers). Unlike IOC$CVTLOGPHY, IOC_STD$CVTLOGPHY does not preserve R3 across the call.

IOC_STD$DELETE_UCB

Deletes the specified UCB if its reference count is zero and UCB$V_DELETEUCB is set in UCB$L_STS.

Module

UCBCREDEL

Format

IOC_STD$DELETE_UCB (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block.

Context

A driver calls IOC_STD$DELETE_UCB with the I/O database locked for write access.

Description

For Digital internal use only.

Macro

CALL_DELETE_UCB

In an Alpha driver, CALL_DELETE_UCB calls IOC_STD$DELETE_UCB using the current contents of R5 as the **ucb** argument.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$DELETE_UCB replaces IOC$DELETE_UCB (used by OpenVMS VAX drivers).

IOC_STD$DIAGBUFILL

Fills a diagnostic buffer if the original $QIO request specified such a buffer.

Module

IOSUBNPAG

Format

IOC_STD$DIAGBUFILL (driver_param, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
driver_param	unspecified	input	reference	required
ucb	UCB	input	reference	required

driver_param
Parameter to be passed to the driver's register dumping routine. Typically, a driver supplies the address of a CRAM in this register.

ucb
Unit control block. IOC_STD$DIAGBUFILL reads the final error retry count from UCB$L_ERTCNT. It obtains the address of the current IRP from UCB$L_IRP and reads the following IRP fields:

Field	Contents
IRP$L_STS	IRP$V_DIAGBUF set if a diagnostic buffer exists
IRP$L_DIAGBUF	Address of diagnostic buffer, if one is present

IOC_STD$DIAGBUFILL obtains the address of the DDB from UCB$L_DDB and the address of the DDT from DDB$L_DDT. The procedure value of driver's register dumping routine is obtained from DDT$L_REGDUMP.

Context

The caller of IOC_STD$DIAGBUFILL may be executing at or above fork IPL and must hold the corresponding fork lock. IOC_STD$DIAGBUFILL returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

A device driver fork process calls IOC_STD$DIAGBUFILL at the end of I/O processing but before releasing the I/O channel. IOC_STD$DIAGBUFILL stores the I/O completion time and the final error retry count in the diagnostic buffer. (IOC_STD$INITIATE has already placed the I/O initiation time [from EXE$GQ_SYSTIME] in the first quadword of the buffer.) IOC_STD$DIAGBUFILL then calls the driver's register dumping routine, passing to it in the **buffer** argument an address within the diagnostic buffer in which the routine can place the register values it retrieves from device interface register space by means of hardware mailbox read transactions. It also passes the contents of the **driver_param** and **ucb** arguments. The register dumping routine fills the remainder of the buffer, and returns to IOC_STD$DIAGBUFILL, which returns to its caller.

Macro

CALL_DIAGBUFILL

In an Alpha driver, the CALL_DIAGBUFILL macro calls IOC_STD$DIAGBUFILL, using the current contents of R4 and R5 as the **driver_parm** and **ucb** arguments, respectively.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$DIAGBUFILL replaces IOC$DIAGBUFILL (used by OpenVMS VAX drivers).
- Prior to calling IOC_STD$DIAGBUFILL, the driver places a parameter, which the routine passes to the driver's register dumping routine, in R4. On OpenVMS Alpha systems, this parameter is often the address of a CRAM (obtained, for instance, from UCB$PS_CRAM or CRB$PS_CRAM). On OpenVMS VAX systems, the parameter similarly would contain the address of the device's CSR.
- The contents of R2 and R3 are destroyed when the caller of IOC_STD$DIAGBUFILL regains control; on OpenVMS VAX systems, these registers contain the DDT address and IRP address respectively.

IOC_STD$FILSPT

Fills a system page-table entry (PTE) with the transfer PTE of a buffer that is locked in memory so that the system PTE may be directly addressed.

Module

BUFFERCTL

Format

sva = IOC_STD$FILSPT (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block. IOC_STD$FILSPT reads UCB$L_SVAPTE to obtain the system virtual address of PTE that maps the first page of the buffer.

Return Values

sva	System virtual address of the first byte in the page that contains the buffer.

Context

The caller of IOC_STD$FILSPT may be executing at fork IPL or above and must hold the corresponding fork lock in a multiprocessing environment.

Description

For Digital internal use only.

Macro

CALL_FILSPT

In an Alpha driver, CALL_FILSPT calls IOC_STD$FILSPT, passing the current contents of R5 as the **ucb** argument. It returns in R0 the system virtual address of the first byte in the page that contains the buffer.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$FILSPT replaces IOC$FILSPT (used by OpenVMS VAX drivers).

IOC_STD$GETBYTE

Fetches a single byte of data from a user buffer.

Module

BUFFERCTL

Format

byte = IOC_STD$GETBYTE (sva, ucb, sva_p)

Arguments

Argument	Type	Access	Mechanism	Status
sva	address	input	reference	required
ucb	UCB	input	reference	required
sva_p	pointer	output	value	required

sva
System virtual address of a single-page window into the user buffer. Prior to calling IOC_STD$GETBYTE, a driver must have called IOC_STD$INITBUFWIND to map the system page-table entry to the user buffer.

ucb
Unit control block. IOC_STD$GETBYTE updates UCB$L_SVAPTE whenever a page boundary is crossed.

sva_p
Location in which IOC_STD$GETBYTE writes the updated system virtual address.

Return Values

byte	One byte of data (not zero-extended) returned from the user buffer.

Context

The caller of IOC_STD$GETBYTE may be executing at fork IPL or above and must hold the corresponding fork lock in a multiprocessing environment.

Description

For Digital internal use only.

Macro

CALL_GETBYTE

In an Alpha driver, CALL_GETBYTE calls IOC_STD$GETBYTE, passing the current contents of R0 and R5 as the **sva** and **ucb** arguments, respectively. It returns in R0 the byte of data (not zero-extended) returned from the user buffer. It returns in R1 the updated system virtual address.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note that IOC_STD$GETBYTE replaces IOC$GETBYTE (used by OpenVMS VAX drivers). Unlike IOC$GETBYTE, IOC_STD$GETBYTE returns the byte of data (and not the updated system virtual address) in R0.

IOC_STD$INITBUFWIND

Initializes a single-page window into a user buffer.

Module

BUFFERCTL

Format

sva = IOC_STD$INITBUFWIND (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block. IOC_STD$INITBUFWIND initializes UCB$L_SVAPTE with the system virtual address of the page-table entry that maps the first page of the buffer.

Return Values

sva	System virtual address of the first byte in the page that contains the buffer.

Context

The caller of IOC_STD$INITBUFWIND may be executing at fork IPL or above and must hold the corresponding fork lock in a multiprocessing environment.

Description

For Digital internal use only.

Macro

CALL_INITBUFWIND

In an Alpha driver, CALL_INITBUFWIND calls IOC_STD$INITBUFWIND, passing the current contents of R5 as the **ucb** argument. It returns in R0 the system virtual address of the first byte in the page that contains the buffer.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$INITBUFWIND replaces IOC$INITBUFWIND (used by OpenVMS VAX drivers).

IOC_STD$INITIATE

Initiates the processing of the next I/O request for a device unit.

Module

IOSUBNPAG

Format

IOC_STD$INITIATE (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp

I/O request packet. IOC_STD$INITIATE reads the following IRP fields:

Field	Contents
IRP$L_SVAPTE	Address of system buffer (buffered I/O) or system virtual address of the PTE that maps process buffer (direct I/O).
IRP$L_BOFF	Byte offset of start of buffer.
IRP$L_BCNT	Size in bytes of transfer.
IRP$W_STS	IRP$V_DIAGBUF set if a diagnostic buffer exists.
IRP$L_DIAGBUF	Address of diagnostic buffer, if one is present. IOC_STD$INITIATE writes the current system time from EXE$GQ_SYSTIME into the first quadword of this buffer.

ucb

Unit control block. IOC_STD$INITIATE reads the following UCB fields:

Field	Contents
UCB$L_DDB	Address of DDB.
UCB$L_DDT	Address of DDT. DDT$PS_START contains the procedure value of the driver’s start-I/O routine.
UCB$L_AFFINITY	Device’s affinity mask.

IOC_STD$INITIATE writes the following UCB fields:

Field	Contents
UCB$L_IRP	Address of IRP
UCB$L_SVAPTE	IRP$L_SVAPTE
UCB$L_BOFF	IRP$L_BOFF
UCB$L_BCNT	IRP$L_BCNT
UCB$L_STS	UCB$V_CANCEL and UCB$V_TIMOUT cleared

Context

IOC_STD$INITIATE is called at fork IPL with the corresponding fork lock held in a multiprocessing system. Within this context, it transfers control to the driver's start-I/O routine.

Description

IOC_STD$INITIATE creates the context in which a driver fork process services an I/O request. IOC_STD$INITIATE creates this context and activates the driver's start-I/O routine in the following steps:

1. Checks the CPU ID of the local processor against the device's affinity mask to determine whether the local processor can initiate the I/O operation on the device. If it cannot, IOC_STD$INITIATE takes steps to initiate the I/O function on another processor in a multiprocessing system. It then returns to its caller.
2. Stores the address of the current IRP in UCB$L_IRP.
3. Copies the transfer parameters contained in the IRP into the UCB:
    a. Copies the address of the system buffer (buffered I/O) or the system virtual address of the PTE that maps process buffer (direct I/O) from IRP$L_SVAPTE to UCB$L_SVAPTE
    b. Copies the byte offset within the page from IRP$L_BOFF to UCB$L_BOFF
    c. Copies the byte count from IRP$L_BCNT to UCB$L_BCNT
4. Clears the cancel-I/O and timeout bits in the UCB status longword (UCB$V_CANCEL and UCB$V_TIMOUT in UCB$L_STS).
5. If the I/O request specifies a diagnostic buffer, as indicated by IRP$V_DIAGBUF in IRP$L_STS, stores the system time in the first quadword of the buffer to which IRP$L_DIAGBUF points (the $QIO system service having already allocated the buffer).
6. Transfers control to the driver's start-I/O routine.

Macro

CALL_INITIATE

In an Alpha driver, the CALL_INITIATE macro calls IOC_STD$INITIATE, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$INITIATE replaces IOC$INITIATE (used by OpenVMS VAX drivers).

IOC_STD$LINK_UCB

Searches the UCB list attached to the device data block identified by the specified UCB and links the specified UCB into the list in ascending unit number order.

Module

UCBCREDEL

Format

status = IOC_STD$LINK_UCB (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block.

Return Values

SS$_NORMAL	Link operation was successful.
SS$_OPINCOMPL	Link operation failed due to the presence of a UCB with the same unit number as the specified UCB.

Context

A driver calls IOC_STD$LINK_UCB with the I/O database locked for write access.

Description

For Digital internal use only.

Macro

CALL_LINK_UCB [interface_warning=YES]

where:

> **interface_warning**=**YES**, the default, specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the VAX version of the corresponding system routine. **interface_warning**=**NO** suppresses the warning.

In an Alpha driver, calls IOC_STD$LINK_UCB using the current contents of R5 as the **ucb** argument. CALL_LINK_UCB returns status in R0 and the address of the newly created UCB in R2, but does not return the address of the UCBs that precede and follow it on the DDB chain in R3 and R1, respectively.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$LINK_UCB replaces IOC$LINK_UCB (used by OpenVMS VAX drivers). IOC_STD$LINK_UCB does not return the addresses of the UCBs that precede and follow the newly-created UCB on the DDB chain.

IOC_STD$MAPVBLK

Maps a virtual block to a logical block using a mapping window.

Module

IOSUBRAMS

Format

status = IOC_STD$MAPVBLK (vbn, numbytes, wcb, irp, ucb, lbn_p, notmapped_p, new_ucb_p)

Arguments

Argument	Type	Access	Mechanism	Status
vbn	integer	input	value	required
numbytes	integer	input	value	required
wcb	WCB	input	reference	required
irp	IRP	input	reference	required
ucb	UCB	input	reference	required
lbn_p	pointer	output	value	required
notmapped_p	pointer	output	value	required
new_ucb_p	pointer	output	value	required

vbn
Virtual block number.

numbytes
Number of bytes to map.

wcb
Window control block.

irp
I/O request packet.

ucb
Unit control block.

lbn_p
Address at which IOC_STD$MAPVBLK writes the logical block number of the first block it maps.

notmapped_p
Address at which IOC_STD$MAPVBLK writes the number of unmapped bytes.

new_ucb_p
Address at which IOC_STD$MAPVBLK writes the address of the updated UCB.

Return Values

status	Low bit set indicates partial map with all output parameters valid, low bit clear indicates total mapping failure with only the **notmapped_p** parameter valid.

Context

IOC_STD$MAPVBLK raises IPL to IPL$_FILSYS and obtains the corresponding spin lock to perform the mapping. As a result, it cannot be called by a driver executing above IPL 8, or by a driver is executing at IPL 8 but holds the IOLOCK8 fork lock.

Description

For Digital internal use only.

Macro

CALL_MAPVBLK

In an Alpha driver, the CALL_MAPVBLK macro calls IOC_STD$MAPVBLK, using the current contents of R0, R1, R2, R3, and R5 as the **vbn**, **numbytes**, **wcb**, **irp** and **ucb** arguments, respectively. It returns status in R0, the address of the logical block number of the first block mapped in R1, the number of unmapped bytes in R2, and the address of the updated UCB in R3. If the low bit of the status value in R0 is clear, signifying failure status, only the value in R2 is valid.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$MAPVBLK replaces IOC$MAPVBLK (used by OpenVMS VAX drivers). Unlike IOC$MAPVBLK, IOC_STD$MAPVBLK does not preserve R3 across the call.

IOC_STD$MNTVER

Assists a driver with mount verification.

Module

IOSUBNPAG

Format

IOC_STD$MNTVER (irp, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required

irp
I/O request packet, or 0. If **irp** contains the address of an IRP, EXE_STD$MNTVER inserts the IRP at the head of the pending-I/O queue in the UCB. If it contains zero, EXE_STD$MNTVER removes the IRP from the head of the pending-I/O queue and attempts to initiate I/O processing.

ucb
Unit control block.

Context

IOC_STD$MNTVER is called at fork IPL with the corresponding fork lock held in a multiprocessing system.

Description

For Digital internal use only.

Macro

CALL_MNTVER

In an Alpha driver, the CALL_MNTVER macro calls IOC_STD$MNTVER, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$MNTVER replaces IOC$MNTVER (used by OpenVMS VAX drivers).

IOC_STD$MOVFRUSER, IOC_STD$MOVFRUSER2

Move data from a user buffer to an internal buffer.

Module

BUFFERCTL

Format

pointer = IOC_STD$MOVFRUSER (sysbuf, numbytes, ucb, sysbuf_p)

pointer = IOC_STD$MOVFRUSER2 (sysbuf, numbytes, ucb, sva, sysbuf_p)

Arguments

Argument	Type	Access	Mechanism	Status
sysbuf	address	input	reference	required
numbytes	integer	input	value	required
ucb	UCB	input	reference	required
sva	address	input	reference	required
sysbuf_p	pointer	output	value	required

sysbuf
Address of internal buffer.

numbytes
Number of bytes to move.

ucb
Unit control block. IOC_STD$MOVFRUSER and IOC_STD$MOVFRUSER2 read the following UCB fields:

Field	Contents
UCB$L_SVAPTE	System virtual address of PTE that maps the first page of the user buffer
UCB$L_SVPN	System virtual page number of SPTE allocated to driver
UCB$L_BOFF	Byte offset within the first page to start of user buffer (IOC_STD$MOVFRUSER only)

sva
System virtual address of the byte in the user buffer after the last byte moved (IOC_STD$MOVFRUSER2 only).

buffptr
System virtual address of the byte in the user buffer after the last byte moved. IOC_STD$MOVFRUSER and IOC_STD$MOVFRUSER2 write this field.

Return Values

pointer	System virtual address of the byte in the internal buffer after the last byte moved.

Context

The caller of IOC_STD$MOVFRUSER or IOC_STD$MOVFRUSER2 may be executing at fork IPL or above and must hold the corresponding fork lock in a multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

A driver calls IOC_STD$MOVFRUSER and IOC_STD$MOVFRUSER2 to move data from a user buffer to a device that cannot itself map the user buffer to system virtual addresses (for instance, a non-DMA device).

To use either routine, the driver must have set bit DPT$V_SVP in the driver prologue table, typically by using the **flags** argument of the DPTAB macro. This causes OpenVMS to allocate a a system page-table entry (SPTE) for driver use. (See the description of the DPTAB macro in Chapter 11 for additional information.)

In order to accomplish the move, IOC_STD$MOVFRUSER and IOC_STD$MOVFRUSER2 first map the user buffer using the system page-table entry (SPTE) the driver allocated in a DPTAB macro invocation. If an SPTE has not been allocated to the driver, these routines cause an access violation when they attempt to refer to the location addressed by the contents of the field UCB$L_SVAPTE.

IOC_STD$MOVFRUSER2 is useful for moving blocks of data in several pieces, each piece beginning within a page rather than on a page boundary. To begin, the driver calls IOC_STD$MOVFRUSER. For each subsequent piece, the driver calls IOC_STD$MOVFRUSER2.

Macro

CALL_MOVFRUSER
CALL_MOVFRUSER2

In an Alpha driver, CALL_MOVFRUSER and CALL_MOVFRUSER2 simulate a JSB to IOC$MOVFRUSER and IOC$MOVFRUSER2 respectively. CALL_MOVFRUSER calls IOC_STD$MOVFRUSER, and CALL_MOVFRUSER2 calls IOC_STD$MOVFRUSER2, passing the current contents of R1, R2, and R5 as the **sysbuf**, **numbytes**, and **ucb** arguments. $MOVFRUSER2 also passes the current contents of R0 as the **sva** argument. Both macros return in R0 and R1, respectively, the system virtual addresses of the bytes in the internal buffer and user buffer after the last byte moved.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$MOVFRUSER and IOC_STD$MOVFRUSER2 replace IOC$MOVFRUSER and IOC$MOVFRUSER2 (used by OpenVMS VAX drivers). Unlike the corresponding OpenVMS VAX routines, both OpenVMS Alpha routines destroy R1 across the call.

IOC_STD$MOVTOUSER, IOC_STD$MOVTOUSER2

Move data from an internal buffer to a user buffer.

Module

BUFFERCTL

Format

pointer = IOC_STD$MOVTOUSER (sysbuf, numbytes, ucb, sysbuf_p)

pointer = IOC_STD$MOVTOUSER2 (sysbuf, numbytes, ucb, sva, sysbuf_p)

Arguments

Argument	Type	Access	Mechanism	Status
sysbuf	address	input	reference	required
numbytes	integer	input	value	required
ucb	UCB	input	reference	required
sva	address	input	reference	required
sysbuf_p	pointer	output	value	required

sysbuf
Address of internal buffer.

numbytes
Number of bytes to move.

ucb
Unit control block. IOC_STD$MOVTOUSER and IOC_STD$MOVTOUSER2 read the following UCB fields:

Field	Contents
UCB$L_SVAPTE	System virtual address of PTE that maps the first page of the user buffer
UCB$L_SVPN	System virtual page number of SPTE allocated to driver
UCB$L_BOFF	Byte offset within the first page to start of user buffer (IOC_STD$MOVTOUSER only)

sva
System virtual address of the byte in the user buffer after the last byte moved (IOC_STD$MOVTOUSER2 only).

buffptr
System virtual address of the byte in the user buffer after the last byte moved. IOC_STD$MOVTOUSER and IOC_STD$MOVTOUSER2 write this field.

Return Values

pointer	System virtual address of the byte in the internal buffer after the last byte moved.

Context

The caller of IOC_STD$MOVTOUSER or IOC_STD$MOVTOUSER2 may be executing at fork IPL or above and must hold the corresponding fork lock in a multiprocessing environment. Either routine returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

A driver calls IOC_STD$MOVTOUSER and IOC_STD$MOVTOUSER2 to move data from a device to a user buffer when the device itself (for instance, a non-DMA device) cannot map the user buffer to system virtual addresses.

To use either routine, the driver must have set bit DPT$V_SVP in the driver prologue table, typically by using the **flags** argument of the DPTAB macro. This causes OpenVMS to allocate a a system page-table entry (SPTE) for driver use. (See the description of the DPTAB macro in Chapter 11 for additional information.)

In order to accomplish the move, IOC_STD$MOVTOUSER and IOC_STD$MOVTOUSER2 first map the user buffer using the system page-table entry (SPTE) the driver allocated in a DPTAB macro invocation. If an SPTE has not been allocated to the driver, these routines cause an access violation when they attempt to refer to the location addressed by the contents of the field UCB$L_SVAPTE.

IOC_STD$MOVTOUSER2 is useful for moving blocks of data in several pieces, each piece beginning within a page rather than on a page boundary. It handles as many pages as you need. To begin, the driver calls IOC_STD$MOVTOUSER. For each subsequent buffer to move, the driver calls IOC_STD$MOVTOUSER2.

Macro

CALL_MOVTOUSER
CALL_MOVTOUSER2

In an Alpha driver, CALL_MOVTOUSER calls IOC_STD$MOVTOUSER, and CALL_MOVTOUSER2 calls IOC_STD$MOVTOUSER2, passing the current contents of R1, R2, and R5 as the **sysbuf**, **numbytes**, and **ucb** arguments. CALL_MOVTOUSER2 also passes the current contents of R0 as the **sva** argument. Both macros return in R0 and R1, respectively, the system virtual addresses of the bytes in the internal buffer and user buffer after the last byte moved.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$MOVTOUSER and IOC_STD$MOVTOUSER2 replace IOC$MOVTOUSER and IOC$MOVTOUSER2 (used by OpenVMS VAX drivers). Unlike the corresponding OpenVMS VAX routines, both OpenVMS Alpha routines destroy R1 across the call.

IOC_STD$PARSDEVNAM

Parses a device name string, checking its syntax and extracting the node name, allocation class number, and unit number.

Module

IOSUBNPAG

Format

status = IOC_STD$PARSDEVNAM (devnamlen, devnam, flags, unit_p, scslen_p, devnamlen_p, devnam_p, flags_p)

Arguments

Argument	Type	Access	Mechanism	Status
devnamlen	integer	input	value	required
devnam	address	input	reference	required
flags	integer	input	value	required
unit_p	pointer	output	reference	required
scslen_p	pointer	output	reference	required
devnamlen_p	pointer	output	reference	required
devnam_p	pointer	output	reference	required
flags_p	pointer	output	reference	required

devnamlen
Size of the name string.

devnam
Name string.

flags
Flags.

unit_p
Address at which IOC_STD$PARSDEVNAM writes an integer representing the unit number.

scslen_p
Address at which IOC_STD$PARSDEVNAM writes an integer representing either the length of the SCS node name, the allocation class number, or the device type code.

devnamlen_p
Address at which IOC_STD$PARSDEVNAM writes an integer representing the size of the name string.

devnam_p

Address at which IOC_STD$PARSDEVNAM writes the address of the name string.

flags_p

Address at which IOC_STD$PARSDEVNAM writes an integer that contains the flags.

Return Values

SS$_IVDEVNAM	Invalid device name string.
SS$_NORMAL	Valid device name string.

Context

IOC_STD$PARSDEVNAM is typically called at fork IPL with the corresponding fork lock held in an OpenVMS multiprocessing system.

Description

For Digital internal use only.

Macro

CALL_PARSDEVNAM

In an Alpha driver, the CALL_PARSDEVNAM macro calls IOC_STD$PARSDEVNAM, using the current contents of R8, R9, and R10 as the **devnamlen**, **devnam**, and **flags** arguments, respectively. When IOC_STD$PARSDEVNAM returns, the macro returns status in R0; the unit number in R2; the length of the SCS node name at the beginning of the name string, allocation class number, or device type code in R3; the size of the name string in R8, the address of the name string in R9, and the flags in R10.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$PARSDEVNAM replaces IOC$PARSDEVNAM (used by OpenVMS VAX drivers). Unlike IOC$PARSDEVNAM, IOC_STD$PARSDEVNAM does not preserve the contents of R8, R9, and R10 across the call.

IOC_STD$POST_IRP

Inserts an I/O request packet in a CPU-specific I/O postprocessing queue.

Module

IOSUBNPAG

Format

IOC_STD$POST_IRP (irp)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required

irp
I/O request block.

Context

Mount verification processing calls IOC_STD$POST_IRP at or above IPL$_ ASTDEL.

Description

For Digital internal use only.

Macro

CALL_POST_IRP

In an Alpha driver, CALL_POST_IRP calls IOC_STD$POST_IRP using the current contents of R3 as the **irp** argument.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$POST_IRP replaces IOC$POST_IRP (used by OpenVMS VAX drivers).

IOC_STD$PTETOPFN

Returns a page frame number (PFN) from a page-table entry (PTE) that has already been determined to be invalid.

Module

BUFFERCTL

Format

pfn = IOC_STD$PTETOPFN (pte)

Arguments

Argument	Type	Access	Mechanism	Status
pte	PTE	input	reference	required

pte
Quadword page-table entry.

Return Values

pfn	Page frame number (zero-extended).

Context

The caller of IOC_STD$PTETOPFN may be executing at or above IPL 0 in kernel mode.

Description

For Digital internal use only.

Macro

CALL_PTETOPFN

In an Alpha driver, CALL_PTETOPFN extracts the quadword page-table entry from R3 and passes a pointer to it as the **pte** argument to IOC_STD$PTETOPFN. It returns the page frame number in R0.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$PTETOPFN replaces IOC$PTETOPFN (used by OpenVMS VAX drivers). Note that, the page-table entry input argument is passed by value (in R3) to IOC$PTETOPFN, but passed by reference to IOC_STD$PTETOPFN.

IOC_STD$QNXTSEG1

Queues the next segment of a virtual I/O request that did not map to a single contiguous I/O request.

Module

IOCIOPOST

Format

IOC_STD$QNXTSEG1 (vbn, bcnt, wcb, irp, pcb, ucb, ucb_p)

Arguments

Argument	Type	Access	Mechanism	Status
vbn	integer	output	value	required
bcnt	integer	output	value	required
wcb	WCB	output	reference	required
irp	IRP	output	reference	required
pcb	PCB	output	reference	required
ucb	UCB	output	reference	required
ucb_p	pointer	input	reference	required

vbn
Virtual block number of the start of the next segment.

bcnt
Required byte count of next segment.

wcb
Window control block.

irp
I/O request packet.

pcb
Process control block.

ucb
Unit control block.

ucb_p
Address at which IOC_STD$QNXTSEG1 writes the address of the unit control block.

Context

The caller of IOC_STD$QNXTSEG1 typically executes at or above fork IPL. IOC_STD$QNXTSEG1 executes at its caller's IPL and returns control at that IPL. The caller retains any spin locks it held at the time of the call.

Description

For Digital internal use only.

Macro

CALL_QNXTSEG1

In an Alpha driver, CALL_QNXTSEG1 calls IOC_STD$QNXTSEG1 using the current contents of R0, R1, R2, R3, R4, and R5 as the **vbn**, **bcnt**, **wcb**, **irp**, **pcb**, and **ucb** arguments. It returns the address of the updated UCB in R5.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$QNXTSEG1 replaces IOC$QNXTSEG1 (used by OpenVMS VAX drivers). Unlike IOC$QNXTSEG1, IOC_STD$QNXTSEG1 does not return the address of the updated UCB in R5.

IOC_STD$PRIMITIVE_REQCHANH, IOC_STD$PRIMITIVE_REQCHANL

Request a controller's data channel and, if unavailable, place process in channel wait queue.

Module

IOSUBNPAG

Format

status = IOC_STD$PRIMITIVE_REQCHANH (irp, ucb, idb_p)

status = IOC_STD$PRIMITIVE_REQCHANL (irp, ucb, idb_p)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
ucb	UCB	input	reference	required
idb_p	pointer	output	reference	required

irp
I/O request packet.

ucb
Unit control block. IOC_STD$PRIMITIVE_REQPCHANH and IOC_STD$PRIMITIVE_REQPCHANL read the following UCB fields:

Field	Contents
UCB$L_FPC	Procedure value of fork routine to be executed when the channel is granted if the channel cannot be granted immediately

Field	Contents
UCB$L_CRB	Address of controller request block (CRB). IOC_STD$REQPCHANH and IOC_STD$REQPCHANL access the following CRB fields:

Field	Contents
CRB$B_MASK	CRB$V_BSY set if the channel is busy
CRB$L_INTD+VEC$L_IDB	Address of IDB
CRB$L_WQFL	Head of queue of UCBs waiting for the controller channel
CRB$L_WQBL	Tail of queue of UCBs waiting for the controller channel

IOC_STD$REQPCHANH and IOC_STD$REQPCHANL write the contents of the **irp** parameter in UCB$Q_FR3, and the address of the UCB in IDB$PS_OWNER.

If the channel is busy, IOC_STD$REQPCHANH and IOC_STD$REQPCHANL update CRB$L_WQFL and CRB$L_WQBL.

idb_p

Address of location in which IOC_STD$REQPCHANH and IOC_STD$REQPCHANL write the address of the interrupt dispatch block (IDB).

Return Values

SS$_NORMAL	Channel has been granted immediately.
0	Channel is busy and UCB fork block has been queued on channel-wait queue.

Context

A driver calls IOC_STD$PRIMITIVE_REQCHANH or IOC_STD$PRIMITIVE_REQCHANL at fork IPL holding the appropriate fork lock. Either IOC_STD$PRIMITIVE_REQCHANH or IOC_STD$PRIMITIVE_REQCHANL, unlike the corresponding OpenVMS VAX system routine, returns to its caller and not to its caller's caller. Each assumes that, prior to the call, its caller has placed the procedure value of the fork routine into UCB$L_FPC.

If the requested channel is busy, either IOC_STD$PRIMITIVE_REQCHANH or IOC_STD$PRIMITIVE_REQCHANL preserves the contents of the **irp** parameter in UCB$Q_FR3 . IOC_STD$RELCHAN eventually calls the fork routine upon granting the channel request, passing the **irp**, **idb**, and **ucb** parameters.

Description

A driver fork process calls IOC_STD$PRIMITIVE_REQCHANH or IOC_STD$PRIMITIVE_REQCHANL to acquire ownership of the controller's data channel.

Each routine examines CRB$V_BSY in CRB$B_MASK. If the selected controller's data channel is idle, the routine grants the channel to the fork process, placing its UCB address in IDB$PS_OWNER and returning successfully with the IDB address in the location specified by the **idb_p** parameter.

If the data channel is busy, the routine saves process context by placing the IRP address, as specified in the **irp** parameter, into the UCB fork block. IOC_STD$REQCHANH then inserts the UCB at the head of the channel wait queue (CRB$L_WQFL); IOC_STD$REQCHANL inserts the UCB at the tail of the queue (CRB$L_WQBL). Finally, the routine returns control to its caller.

When the controller channel is available to a waiting fork process, IOC_STD$RELCHAN resumes the suspended fork process at its channel grant routine, passing to it the **irp**, **idb**, and **ucb** parameters.

Macro

REQCHAN

REQPCHAN

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note that the OpenVMS VAX routines IOC$REQPCHAN and IOC$REQPCHANL are not provided on OpenVMS Alpha systems.

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.

Module

IOSUBNPAG

Format

IOC_STD$PRIMITIVE_WFIKPCH (irp, fr4, ucb, tmo, restore_ipl)

IOC_STD$PRIMITIVE_WFIRLCH (irp, fr4, ucb, tmo, restore_ipl)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
fr4	int64	input	value	required
ucb	UCB	input	reference	required
tmo	integer	input	value	required
restore_ipl	int	input	value	required

irp
I/O request packet.

fr4
Parameter to be passed to the interrupt service routine or timeout handling routine.

ucb
Unit control block. IOC_STD$PRIMITIVE_WFIKPCH and IOC_STD$PRIMITIVE_WFIRLCH read the following UCB fields:

Field	Contents
UCB$L_FPC	Procedure value of fork routine which may be the destination of a JSB instruction issued by either the driver’s interrupt service routine or EXE$TIMEOUT
UCB$B_FLCK	Fork lock index

IOC_STD$PRIMITIVE_WFIKPCH and IOC_STD$PRIMITIVE_WFIRLCH write the following UCB fields:

Field	Contents
UCB$L_DUETIM	Sum of timeout value and EXE$GL_ABSTIM

Field	Contents
UCB$L_STS	UCB$V_INT is set to indicate that interrupts are expected on the device; UCB$V_TIM is set to indicate device I/O is being timed; and UCB$V_TIMOUT is cleared to indicate that unit has not yet timed out.
UCB$Q_FR3	R3 of caller
UCB$Q_FR4	R4 of caller

tmo
Timeout value in seconds.

restore_ipl
IPL to which to lower before returning to caller. This IPL must be the fork IPL associated with device processing and at which the driver was executing prior to invoking the DEVICELOCK macro.

Context

When it is called, IOC_STD$PRIMITIVE_WFIKPCH or IOC_STD$PRIMITIVE_WFIRLCH assumes that the local processor has obtained the appropriate synchronization with the device database by securing the appropriate device lock, as recorded in the unit control block (UCB$L_DLCK) of the device unit from which the interrupt is expected. This requirement also presumes that the local processor is executing at the device IPL associated with the lock.

Before exiting, IOC_STD$PRIMITIVE_WFIKPCH or IOC_STD$PRIMITIVE_WFIRLCH conditionally releases the device lock, so that if the caller of the driver fork thread (the caller's caller) previously owned the device lock, it will continue to hold it when it regains control. IOC_STD$PRIMITIVE_WFIKPCH or IOC_STD$PRIMITIVE_WFIRLCH also lowers the local processor's IPL to the IPL specified in the **restore_ipl** parameter.

Description

A driver fork process calls IOC_STD$PRIMITIVE_WFIKPCH to wait for an interrupt while keeping ownership of the controller's data channel; IOC_STD$PRIMITIVE_WFIRLCH, by contrast, releases the channel.

Either routine performs the following operations:

1. Moves contents of the **irp** and **fr4** parameters into the UCB fork block.
2. Sets UCB$V_INT to indicate an expected interrupt from the device unit.
3. Sets UCB$V_TIM to indicate that OpenVMS should check for timeouts from the device unit.
4. Determines the timeout due time by adding the timeout value specified in R1 to EXE$GL_ABSTIM and storing the result in UCB$L_DUETIM.
5. Clears UCB$V_TIMOUT to indicate that the unit has not yet timed out.
6. Invokes the DEVICEUNLOCK macro to conditionally release the device lock associated with the device unit and to lower IPL to the IPL specified in the **restore_ipl** parameter. These actions presume that the DEVICELOCK macro has been issued prior to the wait-for-interrupt invocation.

7. Returns to its caller.

Note that IOC_STD$PRIMITIVE_WFIRLCH exits by transferring control to IOC_STD$RELCHAN. IOC_STD$RELCHAN releases the controller data channel and eventually issues an RSB instruction to IOC_STD$PRIMITIVE_WFIRLCH which returns to its caller. Because the release of the channel occurs at fork IPL, an interrupt service routine cannot reliably distinguish between operations initiated by IOC_STD$PRIMITIVE_WFIKPCH and IOC_STD$PRIMITIVE_WFIRLCH by examining the ownership of the CRB.

Macro

WFIKPCH

WFIRLCH

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note that the OpenVMS VAX routines IOC$WFIKPCH and IOC$WFIRLCH are not provided on OpenVMS Alpha systems.

IOC_STD$RELCHAN

Releases device ownership of all controller data channels.

Module

IOSUBNPAG

Format

IOC_STD$RELCHAN (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb

Unit control block. IOC_STD$RELCHAN reads UCB$L_CRB to obtain the address of the controller request block (CRB) in order to access the following CRB fields:

Field	Contents
CRB$B_MASK	CRB$V_BSY set if the channel is busy. IOC_STD$RELCHAN clears this bit if no driver is waiting for the controller channel.
CRB$L_INTD+VEC$L_IDB	Address of IDB. IOC_STD$RELCHAN obtains the address the UCB that owns the controller channel from IDB$L_OWNER. IOC_STD$RELCHAN clears IDB$L_OWNER if no driver is waiting for the controller channel.
CRB$L_WQFL	Head of queue of UCBs waiting for the controller.

Context

A driver fork process calls IOC_STD$RELCHAN at fork IPL, holding the corresponding fork lock in a multiprocessing environment. IOC_STD$RELCHAN returns control to its caller after resuming execution of other fork processes waiting for a controller channel.

Description

A driver fork process calls IOC_STD$RELCHAN to release all controller data channels assigned to a device.

If the channel wait queue contains waiting fork processes, IOC_STD$RELCHAN dequeues a process, assigns the channel to that process and calls the suspended fork process at its channel grant routine, passing to it the **irp**, **idb**, and **ucb** parameters.

Macro

CALL_RELCHAN

In an Alpha driver, CALL_RELCHAN calls IOC_STD$RELCHAN using the current contents of R5 as the **ucb** argument.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$RELCHAN replaces IOC$RELCHAN (used by OpenVMS VAX drivers).
- IOC$RELCHAN resumes the fork routine with the address of a device's controller and status register (CSR) in R4. Because OpenVMS Alpha device drivers access device CSRs by means of a controller register access mailbox (CRAM), IOC_STD$RELCHAN provides the IDB address as input to the reactivated fork routine. The fork routine uses the IDB address as input to the driver macros and routines that manipulate CSRs by means of the CRAM.

IOC_STD$REQCOM

Completes an I/O operation on a device unit, requests I/O postprocessing of the current request, and starts the next I/O request waiting for the device.

Module

IOSUBNPAG

Format

IOC_STD$REQCOM (iost1, iost2, ucb)

Arguments

Argument	Type	Access	Mechanism	Status
iost1	integer	input	value	required
iost2	integer	input	value	required
ucb	UCB	input	reference	required

iost1
First longword of I/O status.

iost2
Second longword of I/O status.

ucb
Unit control block. IOC_STD$REQCOM accesses the following UCB fields:

Field	Contents
UCB$L_ERTCNT	Final error count.
UCB$L_ERTMAX	Maximum error retry count.
UCB$L_EMB	Address of error message buffer.
UCB$L_IRP	Address of IRP. IOC_STD$REQCOM writes **iost1** and **iost2** into IRP$L_IOST1 and IRP$L_IOST2, respectively.
UCB$B_DEVCLASS	DC$_DISK and DC$_TAPE devices are subject to mount verification checks.
UCB$L_IOQFL	Device unit's pending-I/O queue. IOC_STD$REQCOM updates this field.

<table>
<tr><th>Field</th><th>Contents</th></tr>
<tr><td>UCB$L_STS</td><td>If error logging is in progress (that is, UCB$V_ERLOGIP is set), IOC_STD$REQCOM writes the following fields in the error message buffer:
<table>
<tr><th>Field</th><th>Contents</th></tr>
<tr><td>EMB$L_DV_STS</td><td>UCB$L_STS.</td></tr>
<tr><td>EMB$L_DV_ERTCNT</td><td>UCB$L_ERTCNT.</td></tr>
<tr><td>EMB$L_DV_ERTCNT+1</td><td>UCB$L_ERTMAX.</td></tr>
<tr><td>EMB$Q_DV_IOSB</td><td>Quadword of I/O status.</td></tr>
</table>
IOC_STD$REQCOM then clears UCB$V_BSY and UCB$V_ERLOGIP.</td></tr>
<tr><td>UCB$L_OPCNT</td><td>Unit operations count. IOC_STD$REQCOM increases this field.</td></tr>
</table>

Context

A driver fork process calls IOC_STD$REQCOM at fork IPL, holding the corresponding fork lock in a multiprocessing environment. IOC_STD$REQCOM transfers control to IOC_STD$RELCHAN, which may call the OpenVMS fork dispatcher to resume another driver fork process. When it regains control, IOC_STD$REQCOM returns to the driver fork process.

Description

A driver fork process calls this routine after a device I/O operation and all device-dependent processing of an I/O request is complete.

IOC$REQCOM performs the following tasks:

1. If error logging is in progress for the device (as indicated by UCB$V_ERLOGIP in UCB$L_STS), writes into the error message buffer the status of the device unit, the error retry count for the transfer, the maximum error retry count for the driver, and the final status of the I/O operation. It then releases the error message buffer by calling ERL_STD$RELEASEMB.
2. Increases the device unit's operations count (UCB$L_OPCNT).
3. If UCB$B_DEVCLASS specifies a disk device (DC$_DISK) or tape device (DC$_TAPE) and error status is reported, performs a set of checks to determine if mount verification is necessary. Tape end-of-file (EOF) errors (SS$_ENDOFFILE) are exempt from these checks. For a tape device with success status, checks to determine if CRC must be generated.
4. Writes final I/O status (R0 and R1) into IRP$L_IOST1 and IRP$L_IOST2.
5. Inserts the IRP in systemwide I/O postprocessing queue.
6. Requests a software interrupt from the local processor at IPL$_IOPOST.
7. Attempts to remove an IRP from the device's pending-I/O queue (at UCB$L_IOQFL). If successful, it transfers control to IOC_STD$INITIATE to begin driver processing of this I/O request. If the queue is empty, it clears the unit busy bit (UCB$V_BSY in UCB$L_STS) to indicate that the device is idle.

8. Exits by transferring control to IOC_STD$RELCHAN.

Macro

CALL_REQCOM

In an Alpha driver, the CALL_REQCOM macro calls IOC_STD$REQCOM, using the current contents of R0, R1, and R5 as the **iost1**, **iost2**, and **ucb** arguments, respectively.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$REQCOM replaces IOC$REQCOM (used by OpenVMS VAX drivers). Unlike IOC$REQCOM, IOC_STD$REQCOM does not return the addresses of the IRP and UCB in R3 and R5, respectively.
- The Alpha REQCOM macro returns control to the driver fork process, which itself must issue the return to its caller.

IOC_STD$SEARCHDEV

Searches the I/O database for a specific physical device.

Module

IOSUBPAGD

Format

status = IOC_STD$SEARCHDEV (descr_p, ucb_p, ddb_p, sb_p)

Arguments

Argument	Type	Access	Mechanism	Status
descr_p	pointer	input	reference	required
ucb_p	pointer	output	reference	required
ddb_p	pointer	output	reference	required
sb_p	pointer	output	reference	required

descr_p
Descriptor of device logical name.

ucb_p
Address at which IOC_STD$SEARCHDEV writes the unit control block (UCB) address.

ddb_p
Address at which IOC_STD$SEARCHDEV writes the device data block (DDB) address.

sb_p
Address at which IOC_STD$SEARCHDEV writes the system block (SB) address.

Return Values

SS$_ACCVIO	Name string is not readable.
SS$_DEVALLOC	Device is allocated to another user.
SS$_DEVMOUNT	Device already mounted.
SS$_DEVOFFLINE	Device marked offline.
SS$_IVDEVNAM	Invalid device name string.
SS$_IVLOGNAM	Invalid logical name.
SS$_NODEVAVL	Device exists but is not available.
SS$_NONLOCAL	Nonlocal device.
SS$_NOPRIV	Insufficient privilege to access device.

SS$_NORMAL	Device found.
SS$_NOSUCHDEV	Device not found.
SS$_TEMPLATEDEV	Cannot allocate template device.
SS$_TOOMANYLNAM	Maximum logical name recursion limit exceeded.

Context

A driver calls IOC_STD$SEARCHDEV at IPL$_ASTDEL holding the I/O database mutex.

Description

For Digital internal use only.

Macro

CALL_SEARCHDEV

In an Alpha driver, the CALL_SEARCHDEV macro calls IOC_STD$SEARCHDEV, using the current contents of R1 as the **descr_p** argument. When IOC_STD$SEARCHDEV returns, the macro returns returns status in R0, the UCB address in R1, the DDB address in R2, and the SB address in R3.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$SEARCHDEV replaces IOC$SEARCHDEV (used by OpenVMS VAX drivers). Unlike IOC$SEARCHDEV, IOC_STD$SEARCHDEV does not provide the addresses of the UCB, DDB, and SB in R1, R2, and R3, respectively.

IOC_STD$SEARCHINT

Searches the I/O database for the specified device, using specified search rules.

Module

IOSUBNPAG

Format

status = IOC_STD$SEARCHINT (unit, scslen, devnamlen, devnam, flags, ucb_p, ddb_p, sb_p, lock_val_p)

Arguments

Argument	Type	Access	Mechanism	Status
unit	integer	input	value	required
scslen	integer	input	value	required
devnamlen	integer	input	value	required
devnam	address	input	reference	required
flags	integer	input	value	required
ucb_p	pointer	output	reference	required
ddb_p	pointer	output	reference	required
sb_p	pointer	output	reference	required
lock_val_p	pointer	output	reference	required

unit
Unit number.

scslen
Integer representing either the length of the SCS node name, the allocation class number, or the device type code.

devnamlen
Size of the name string.

devnam
Name string.

flags
Flags.

ucb_p
Address at which IOC_STD$SEARCHINT writes the UCB address.

ddb_p
Address at which IOC_STD$SEARCHINT writes the DDB address.

sb_p
Address at which IOC_STD$SEARCHINT writes the system block (SB) address.

lock_val_p
Address at which IOC_STD$SEARCHINT writes the address of the lock value block.

Return Values

SS$_DEVMOUNT	Device already mounted.
SS$_DEVOFFLINE	Device marked offline.
SS$_NODEVAVL	Device exists but is not available.
SS$_NOPRIV	Insufficient privilege to access device.
SS$_NORMAL	Device found.
SS$_NOSUCHDEV	Device not found.
SS$_TEMPLATEDEV	Cannot allocate template device.

Context

A driver calls IOC_STD$SEARCHINT at IPL$_ASTDEL holding the I/O database mutex. It may be called at elevated IPL only for searches specifying IOC$V_ANY.

Description

For Digital internal use only.

Macro

CALL_SEARCHINT

In an Alpha driver, the CALL_SEARCHINT macro calls IOC_STD$SEARCHINT, using the current contents of R2, R3, R8, R9 and R10 as the **unit**, **scslen**, **devnamlen**, **devnam**, and **flags** arguments, respectively. When IOC_STD$SEARCHINT returns, the macro returns status in R0, the UCB address in R5, the DDB address in R6, and the SB address in R7.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$SEARCHINT replaces IOC$SEARCHINT (used by OpenVMS VAX drivers). Unlike IOC$SEARCHINT, IOC_STD$SEARCHINT does not provide the addresses of the UCB, DDB, and SB in R5, R6, and R7, respectively.

IOC_STD$SENSEDISK

Copies the disk's size in logical blocks from the device's UCB into the second longword of the I/O status block (IOSB) specified in a $QIO system service call, and completes the I/O operation successfully.

Module

IOSUBRAMS

Format

status = IOC_STD$SENSEDISK (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet for the current I/O request.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_NORMAL	The routine completed successfully.

Context

FDT dispatching code in the $QIO system service calls IOC_STD$SENSEDISK as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine IOC$SENSEDISK (used by OpenVMS VAX device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to IOC_STD$SENSEDISK.
- IOC_STD$SENSEDISK returns control to the system service dispatcher, passing it the final $QIO system service status (SS$_NORMAL) in R0. IOC_STD$SENSEDISK returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

IOC_STD$SEVER_UCB

Removes the specified UCB from the UCB list of the device data block identified within the specified UCB.

Module

UCBCREDEL

Format

IOC_STD$SEVER_UCB (ucb)

Arguments

Argument	Type	Access	Mechanism	Status
ucb	UCB	input	reference	required

ucb
Unit control block.

Context

A driver calls IOC_STD$SEVER_UCB with the I/O database locked for write access.

Description

For Digital internal use only.

Macro

CALL_SEVER_UCB

In an Alpha driver, CALL_SEVER_UCB calls IOC_STD$SEVER_UCB using the current contents of R5 as the **ucb** argument.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$SEVER_UCB replaces IOC$SEVER_UCB (used by OpenVMS VAX drivers).

IOC_STD$SIMREQCOM

Completes an I/O operation by setting an event flag, modifying an I/O status block (IOSB), setting an event flag, or queuing an AST to the process requesting the I/O. The caller of this routine is responsible for checking quotas and updating the I/O count.

Module

IOCIOPOST

Format

status = IOC_STD$SIMREQCOM (iosb, pri, efn, iost, acb, acmode)

Arguments

Argument	Type	Access	Mechanism	Status
iosb	IOSB	input	reference	optional
pri	integer	input	value	optional
efn	integer	input	value	optional
iost	unspecified	input	unspecified	required
acb	ACB	input	reference	optional
acmode	integer	input	value	optional

iosb

I/O status block. If this parameter contains the address of an IOSB, IOC_STD$SIMREQCOM checks for write access to the IOSB. If it contains a zero, IOC_STD$SIMREQCOM makes no IOSB modifications.

pri

Priority boost class to be passed directly to SCH$POSTEF and SCH$QAST. If an IOSB address is supplied to the **iosb** parameter, this parameter has no effect. If this parameter contains a zero, there is no priority boost.

efn

Common or local event flag to be set. If this parameter contains –1, no event flag is set.

iost

Internal process identification (IPID) of the target process (if the **iosb** parameter is zero); address of a quadword containing the new contents of the user's IOSB (if the **iosb** is non-zero).

acb

AST control block. If this parameter is zero, no AST is delivered. When the **acb** parameter is non-zero and ACB$L_AST is zero, IOC_STD$SIMREQCOM checks ACB$V_NODELETE. If ACB$V_NODELETE is clear, IOC_STD$SIMREQCOM uses ACB$W_SIZE to return the ACB and any structure in which it is embedded to nonpaged pool.

acmode
Access mode of the process originally requesting the I/O operation. IOC_STD$SIMREQCOM uses this value to probe the IOSB (if specified) for write access. If the **iosb** parameter is zero, this parameter is ignored.

Return Values

SS$_ILLEFC	Illegal cluster number.
SS$_NONEXPR	Nonexistent process.
SS$_NORMAL	Normal, successful completion.
SS$_UNASEFC	Unassigned cluster number.
SS$_WASCLR	Specified event flag was clear initially.
SS$_WASSET	Specified event flag was set initially.

Context

If supplying a non-zero value for the **iosb** parameter, the caller of IOC_STD$SIMREQCOM must be executing in the context of the target process.

Description

For Digital internal use only.

Macro

CALL_SIMREQCOM

In an Alpha driver, the CALL_SIMREQCOM macro calls IOC_STD$SIMREQCOM, using the current contents of R1, R2, R3, R4, R5, and R6 as the **iosb**, **pri**, **efn**, **iost**, **acb**, and **acmode** arguments, respectively.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$SIMREQCOM replaces IOC$SIMREQCOM (used by OpenVMS VAX drivers).

IOC_STD$THREADCRB

Threads a controller request block (CRB) onto the CRB timeout queue chain headed by IOC$GL_CRBTMOUT.

Module

IOSUBNPAG

Format

IOC_STD$THREADCRB (crb)

Arguments

Argument	Type	Access	Mechanism	Status
crb	CRB	input	reference	required

crb
Controller request block.

Context

Mount verification processing calls IOC_STD$THREADCRB at or above IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

CALL_THREADCRB [save_r0]

where:

> **save_r0** indicates that the macro should preserve register R0 across the call to IOC_STD$THREADCRB. If **save_r0** is blank or **save_r0**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r0**=**NO**, R0 is not saved.)

In an Alpha driver, CALL_THREADCRB calls IOC_STD$THREADCRB using the current contents of R3 as the **crb** argument. Unless you specify **save_r1**=**NO**, the macro preserves the quadword register R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- IOC_STD$THREADCRB replaces IOC$THREADCRB (used by OpenVMS VAX drivers). Unlike IOC$THREADCRB, IOC_STD$THREADCRB routines does not preserve R0 across the call.

MMG_STD$IOLOCK

Locks process pages in memory.

Module

IOLOCK

Format

status = MMG_STD$IOLOCK (buf, bufsize, is_read, pcb, svapte_p)

Arguments

Argument	Type	Access	Mechanism	Status
buf	address	input	reference	required
bufsize	integer	input	value	required
is_read	integer	input	value	required
pcb	PCB	input	reference	required
svapte_p	pointer	output	reference	required

buf
Buffer.

bufsize
Size of output buffer in bytes.

is_read
Transfer direction indicator, as follows:

Value	Description
0	Write from memory to I/O device
1	Read into memory from I/O device
5	Write from and read into memory from I/O device

pcb
Process control block.

svapte_p
Address of location in which MMG_STD$IOLOCK returns either the system virtual address of the first page-table entry (if the returned status is SS$_NORMAL) or the address of a page to be faulted into memory (if the returned status is 0).

Return Values

SS$_ACCVIO	Specified buffer is not a process buffer, but does not fully reside in system space; or process buffer overruns balance set slots.
SS$_INSFWSL	Insufficient working set list.
SS$_NORMAL	Normal, successful completion.
0	Virtual address must be faulted into memory.

Context

MMG_STD$IOLOCK must be called at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

CALL_IOLOCK

In an Alpha driver, CALL_IOLOCK calls MMG_STD$IOLOCK using the current contents of R0, R1, R2, and R4 as the **buf**, **bufsize**, **is_read**, and **pcb** arguments, respectively.

CALL_IOLOCK returns status in R0. If R0 contains SS$_NORMAL, R1 contains the system virtual address of the first page-table entry. If R0 contains zero, R1 contains the address of a page to be faulted into memory. R0 can also contain a system-level status.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- MMG_STD$IOLOCK replaces MMG$IOLOCK (used by OpenVMS VAX drivers).

MMG_STD$UNLOCK

Unlocks process pages previously locked for a direct-I/O operation.

Module

IOLOCK

Format

MMG_STD$UNLOCK (npages, svapte)

Arguments

Argument	Type	Access	Mechanism	Status
npages	integer	input	value	required
svapte	integer	input	value	required

npages
Number of buffer pages to unlock.

svapte
System virtual address of PTE for the first buffer page.

Context

Because MMG_STD$UNLOCK raises IPL to IPL$_SYNCH, and obtains the MMG spin lock in a multiprocessing environment, its caller cannot be executing above IPL$_SYNCH or hold any higher ranked spin locks. MMG_STD$UNLOCK returns control to its caller at the caller's IPL. The caller retains any spin locks it held at the time of the call.

Description

Drivers rarely use MMG_STD$UNLOCK. At the completion of a direct-I/O transfer, IOC_STD$IOPOST automatically unlocks the pages of both the user buffer and any additional buffers specified in region 1 (if defined) and region 2 (if defined) for all the IRPEs linked to the packet undergoing completion processing.

However, driver FDT routines do use MMG_STD$UNLOCK when an attempt to lock IRPE buffers for a direct-I/O transfer fails. The buffer-locking routines called by such a driver (EXE_STD$READLOCK, EXE_STD$WRITELOCK, and EXE_STD$MODIFYLOCK) allow a driver to specify an error-handling callback routine that can call MMG_STD$UNLOCK to unlock all previously locked regions and deallocate the IRPE using EXE_STD$DEANONPAGED.

Macro

CALL_UNLOCK

In an Alpha driver, CALL_UNLOCK calls MMG_STD$UNLOCK using the current contents of R1 and R3 as the **npages** and **svapte** arguments, respectively.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- MMG_STD$UNLOCK replaces MMG$UNLOCK (used by OpenVMS VAX drivers).

MT_STD$CHECK_ACCESS

Checks access rights for magtape control write functions.

Module

MTFDT

Format

status = MT_STD$CHECK_ACCESS (irp, pcb, ucb, ccb)

Arguments

Argument	Type	Access	Mechanism	Status
irp	IRP	input	reference	required
pcb	PCB	input	reference	required
ucb	UCB	input	reference	required
ccb	CCB	input	reference	required

irp
I/O request packet.

pcb
Process control block of the current process.

ucb
Unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.

ccb
Channel control block that describes the process-I/O channel.

Return Values

SS$_FDT_COMPL	Warning-level status indicating that FDT processing is complete. The routine that receives this status can no longer safely access the IRP.

Status in FDT_CONTEXT

SS$_ACCVIO	Process does not have write access to volume.
SS$_NORMAL	I/O request has been successfully queued to the driver's start-I/O routine.
SS$_NOPRIV	Process has insufficient privileges to perform a control write function.
SS$_WRITLCK	Device software is write locked.

Context

FDT dispatching code in the $QIO system service calls MT_STD$CHECK_ACCESS as an upper-level FDT action routine at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

None.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The FDT routine MT$CHECK_ACCESS (used by OpenVMS VAX device drivers) expects as input in R7 the number of the bit that specifies the code for the requested I/O function from R7, and, in R8, the address of the entry in the function decision table from which it received control.

  R0, R7, and R8 are not provided as input to MT_STD$CHECK_ACCESS.
- Upon a successful return from MT$CHECK_ACCESS, its OpenVMS VAX callers needed to call EXE$ZEROPARM to queue the request to the driver's start-I/O routine.

  If the volume is not write-locked and the requesting process has write access to the volume. MT_STD$CHECK_ACCESS automatically invokes the CALL_QIODRVPKT macro.
- MT$CHECK_ACCESS returns control to the system service dispatcher, passing it the final $QIO system service status in R0. MT_STD$CHECK_ACCESS returns to its caller, passing it SS$_FDT_COMPL status in R0 and storing the final $QIO system service status in the FDT_CONTEXT structure. The $QIO system service retrieves the status from this structure.

SCH_STD$IOLOCKR

Locks the I/O database mutex on behalf of its caller for read access.

Module

MUTEX

Format

pointer = SCH_STD$IOLOCKR (pcb)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required

pcb
Process control block.

Return Values

pointer	Address of I/O database mutex.

Context

SCH_STD$IOLOCKR must be called at or below IPL$_SYNCH. It returns to its caller at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

CALL_IOLOCKR [save_r1]

where:

> **save_r1** indicates that the macro should preserve register R1 across the call to SCH_STD$IOLOCKR. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

In an Alpha driver, CALL_IOLOCKR calls SCH_STD$IOLOCKR using the current contents of R4 as the **pcb** argument.

CALL_IOLOCKR returns the address of the I/O database mutex in R0. Unless you specify **save_r1**=**NO**, the macro preserves R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- SCH_STD$IOLOCKR replaces SCH$IOLOCKR (used by OpenVMS VAX drivers). Unlike SCH$IOLOCKR, SCH_STD$IOLOCKR destroys the contents of R1 through R3 across the call.

SCH_STD$IOLOCKW

Locks the I/O database mutex on behalf of its caller for write access.

Module

MUTEX

Format

pointer = SCH_STD$IOLOCKW (pcb)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required

pcb
Process control block.

Return Values

pointer	Address of I/O database mutex.

Context

SCH_STD$IOLOCKW must be called at or below IPL$_SYNCH. It returns to its caller at IPL$_ASTDEL.

Description

For Digital internal use only.

Macro

CALL_IOLOCKW [save_r1]

where:

> **save_r1** indicates that the macro should preserve register R1 across the call to SCH_STD$IOLOCKW. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

In an Alpha driver, CALL_IOLOCKW calls SCH_STD$IOLOCKW using the current contents of R4 as the **pcb** argument.

CALL_IOLOCKW returns the address of the I/O database mutex in R0. Unless you specify **save_r1**=**NO**, the macro preserves R1 across the call.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- SCH_STD$IOLOCKW replaces SCH$IOLOCKW (used by OpenVMS VAX drivers). Unlike SCH$IOLOCKW, SCH_STD$IOLOCKW destroys the contents of R1 through R3 across the call.

SCH_STD$IOUNLOCK

Releases ownership of the I/O database mutex and, if the mutex has thus become available to waiting processes, reactivates the next eligible process.

Module

MUTEX

Format

SCH_STD$IOUNLOCK (pcb)

Arguments

Argument	Type	Access	Mechanism	Status
pcb	PCB	input	reference	required

pcb
Process control block.

Context

SCH_STD$IOUNLOCK must be called below IPL$_SCHED. It returns to its caller at its caller's IPL.

Description

For Digital internal use only.

Macro

CALL_IOUNLOCK

In an Alpha driver, CALL_IOUNLOCK calls SCH_STD$IOUNLOCK using the current contents of R4 as the **pcb** argument.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- SCH_STD$IOUNLOCK replaces SCH$IOUNLOCK (used by OpenVMS VAX drivers). Unlike SCH$IOUNLOCK, SCH_STD$IOUNLOCK destroys the contents of R1 through R3 across the call.

10

Data Structures

Because a driver and the operating system cooperate to process an I/O request, they must have a common and current source of information about the request and the components of the I/O subsystem involved in servicing the request. This information source consists of a set of data structures collectively known as the **I/O database**.

Components of the I/O database include the following:

- Structures that describe individual hardware components, such as devices, controllers, adapters, and widgets. In this category are the following:

Structure	Description	Associated Structures
Unit control block (UCB)	Records the current status of an I/O device unit attached to the OpenVMS system	Object rights block (ORB), Controller register access mailbox (CRAM), Fork block (FKB)
Device data block (DDB)	Describes the common characteristics of devices of the same type connected to a particular controller	—
Channel request block (CRB)	Describes the current state of an I/O controller	Interrupt transfer vector block (VEC), Fork block (FKB)
Interrupt dispatch block (IDB)	Provides information that supplements that contained in the CRB, enabling the system to correctly dispatch and service interrupts from a device unit attached to a controller	Vector list extension (VLE), Controller register access mailbox (CRAM)
Adapter control block (ADP)	Describes the processor-memory interconnect (PMI), a tightly coupled I/O interconnect, or a multichannel I/O widget	Adapter bus array (BUSARRAY)

- Driver tables that allow the system to load drivers, validate device functions, and call driver routines at their entry points. In this category are the following:

Structure	Description	Associated Structures
Driver prologue table (DPT)	Contains information that allows the driver-loading procedure to load the driver into memory and initialize the I/O database according to the number and type of devices supported by the driver	—
Driver dispatch table (DDT)	Contains procedure values representing all external driver entry points (with the exception of the interrupt service routine) and the address of the driver's function decision table (FDT)	—
Function decision table (FDT)	Identifies those I/O functions supported by a device and associates valid function codes with the addresses of I/O preprocessing routines (also known as FDT routines)	—

- Structures that describe the context of a request for I/O activity. In this category are the following:

Structure	Description	Associated Structures
Channel control block (CCB)	Describes the software I/O channel that links a process to the target device of an I/O operation	—
I/O request packet (IRP)	Describes a pending or in-progress I/O request	I/O request packet extension (IRPE)

- Miscellaneous structures, such as the following:

Structure	Description	Associated Structures
Kernel process block (KPB)	Describes the scheduling and suspension mechanisms associated with a kernel process and records its suspended context	Fork block (FKB)
Counted resource allocation block (CRAB)	Records the number and type of a counted shared resource, such as a set of map registers, available to drivers	Counted resource context block (CRCTX)
Controller register access mailbox (CRAM)	Describes a read or write transaction to device interface register space	—

Figure 10–1 shows the relationships among the principal data structures in the I/O database.

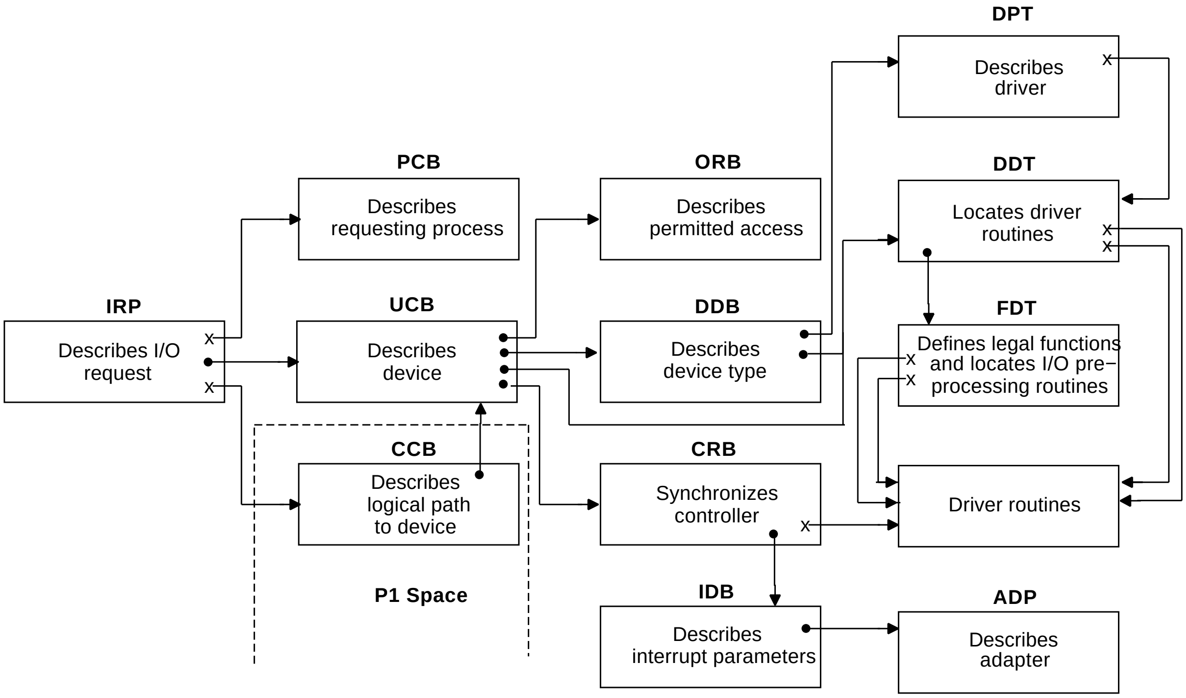
This chapter describes those structures referenced by driver code. It lists their fields in the order in which they appear in the structures. All data structures discussed in this chapter, with the exception of the channel control block (CCB), exist in nonpaged system memory.

Notes

Fields marked "Reserved" or "Unused" are reserved by Digital unless otherwise specified.

When referring to locations within a data structure, a driver should use symbolic offsets, not numeric offsets, from the beginning of the structure. Numeric offsets are likely to change with each new release of the OpenVMS operating system. The figures in this chapter list OpenVMS Alpha Version 6.1 numeric offsets to aid in driver debugging.

Figure 10–1 I/O Database



10.1 ADP (Adapter Control Block)

An adapter control block (ADP) represents a hardware block that connects one interconnect to another. OpenVMS Alpha I/O configuration code creates an ADP for the processor-memory interconnect (PMI), each tightly coupled I/O interconnect, and each multichannel I/O widget.

The system ADP represents the PMI. Any other ADP represents either a tightly coupled I/O interconnect or a multichannel I/O widget.

- An ADP for a tightly coupled I/O interconnect contains information related to hardware mailbox support, system topology, adapter interrupts, and related items. It also contains information about the I/O adapter that connects the interconnect to the PMI or to a parent tightly coupled I/O interconnect. The adjective **parent** in this context describes the tightly coupled I/O interconnect that is closer to the PMI.
- Although information relating to an I/O widget is normally maintained only in a widget-specific data structure defined and used by the widget's driver, information that is common to all loosely coupled I/O interconnects that connect to a multichannel I/O widget is maintained in an ADP.

Table 10–1 defines the fields that appear in an ADP. Bus-specific extensions start at offset ADP$L_XBIA_CSR in the ADP.

An ADP can have up to four auxiliary data structures:

- An adapter bus array (BUSARRAY), pointed to by ADP$PS_BUS_ARRAY
- An adapter command table (CMDTABLE), pointed to by ADP$PS_COMMAND_TBL
- A counted resource allocation block (CRAB), pointed to by ADP$L_CRAB
- An indirect interrupt vector dispatch table, pointed to by ADP$L_VECTOR

IOC$GL_ADPLIST is the listhead for the list of all ADPs in the system. The first ADP in the ADP list is the system ADP. Offset ADP$L_LINK in each ADP points to the next ADP in this list. The last ADP in the list contains a zero in this field. The SYSMAN command IO SHOW ADAPTER traverses this list and displays its contents.

The hierarchy of tightly coupled I/O interconnects in a system is represented by the interconnection between the ADPs in the ADP list. In conjunction with the auxiliary BUSARRAY structure of each ADP, this information represents a system's configuration.

At the root of the hierarchical ADP list is the system ADP. Offset ADP$PS_CHILD_ADP in the system ADP points to an ADP for a tightly coupled I/O interconnect at the next level in the hierarchy — one that connects to the PMI directly: that is, without other intervening interconnects.

Offset ADP$PS_PEER_ADP in the system ADP is always zero because the system ADP has no peers. The DEC 4000 Alpha system local bus (L-bus) and Futurebus+ are both tightly coupled I/O interconnects that are directly connected to the C-bus through the DEC 4000 Alpha system I/O module. Offset ADP$PS_PEER_ADP in the L-bus ADP points to the Futurebus+ ADP, because the Futurebus+ is the L-bus's peer, connecting to the system at the same level as the L-bus. ADP$PS_CHILD_ADP in each of the L-bus and Futurebus+ ADPs contains a zero.

Table 10–1 Contents of Adapter Control Block

Field	Use
ADP$Q_CSR	Address of adapter control and status register (CSR), which marks the base of adapter register space on the remote tightly coupled I/O interconnect. This may be either a virtual or physical address, depending upon the adapter. The OpenVMS adapter initialization routine writes this field. The IOC$CRAM_CMD uses the CSR address in calculations that set up driver transactions to and from remote adapter I/O space by means of hardware I/O mailboxes. For single-channel adapters, the contents of ADP$Q_CSR and IDB$Q_CSR are often the same. For multichannel adapters, ADP$Q_CSR contains the base address of the common adapter register space, and individual IDBs point to the specific adapter registers associated with individual channels.
ADP$W_SIZE	Size of ADP in bytes. Depending upon the type of I/O adapter being described, the ADP size is variable and subject to the length of the bus-specific ADP extension. The OpenVMS adapter initialization routine writes this field when the routine creates the ADP.
ADP$B_TYPE	Type of data structure. The OpenVMS adapter initialization routine writes the symbolic constant DYN$C_ADP into this field when the routine creates the ADP.
ADP$B_NUMBER	Number of this type of adapter. This field is currently unused in OpenVMS Alpha systems.
ADP$L_LINK	Pointer to the next ADP in the ADP list (headed by IOC$GL_ADPLIST). The last ADP in the list contains a zero in this field.
ADP$L_TR	Nexus number of adapter. The OpenVMS adapter initialization routine assigns a nexus number to each node it encounters as it probes an I/O interconnect. When processing an SYSMAN IO CONNECT command which specifies the /ADAPTER qualifier the driver-loading procedure compares the specified nexus number with this field of each ADP in the system to locate the adapter to which the device serviced by the driver is attached.
ADP$L_ADPTYPE	Type of ADP. The OpenVMS adapter initialization routine writes a symbolic constant (defined by the $DCDEF macro in SYS$LIBRARY:STARLET.MLB) into this field when the routine creates an ADP.

(continued on next page)

Table 10–1 (Cont.) Contents of Adapter Control Block

Field	Use
ADP$L_VECTOR	Address of indirect interrupt vector dispatch table. For adapters that service indirect interrupts, the OpenVMS adapter initialization routine sets ADP$V_INDIRECT_VECTOR in ADP$L_ADAPTER_FLAGS, and allocates sufficient nonpaged dynamic memory for this table. Each entry in this table consists of a longword pointer to the VEC substructure of a CRB of a device for which the system dispatches interrupts through this ADP. ADPs that service indirectly-vectored device interrupts include a VEC substructure at ADP$L_INTD (as described in Section 10.5) that contains the code address (VEC$PS_ISR_CODE), procedure descriptor address (VEC$PS_ISR_PD), and parameter field (VEC$L_IDB, which contains the address of the ADP) of the adapter's indirect interrupt service routine. The SCB entries assigned to devices that interrupt indirectly contain the code address of the common interrupt dispatcher and, as the parameter, the address of ADP$L_INTD. The common interrupt dispatcher issues a standard call to the ADP's indirect interrupt service routine, which determines the interrupt vector of the interrupting device, using it as an index into the indirect interrupt vector dispatch table. The ADP's indirect interrupt service routine thereby locates the appropriate device driver's interrupt service routine and calls it, passing it the address of the IDB as the only parameter.
ADP$L_CRB	Address of controller request block (CRB) associated with the ADP. In the case of an ADP that describes a multichannel I/O widget, this field represents the head of a singly-linked list of CRBs linked together by the field CRB$PS_CRB_LINK.
ADP$PS_MBPR	Virtual address of mailbox pointer register (MBPR). The OpenVMS adapter initialization routine initializes this field.
ADP$Q_QUEUE_TIME	Timeout value for mailbox queuing operation. The OpenVMS adapter initialization routine initializes this field with the number of nanoseconds it takes to write the physical address of a hardware I/O mailbox to the MBPR without a timeout occurring.
ADP$Q_WAIT_TIME	Timeout value for the completion of a hardware I/O mailbox transaction. The OpenVMS adapter initialization routine initializes this field with the number of nanoseconds a thread should wait, before timing out, for the hardware I/O mailbox DON bit to be set.
ADP$PS_PARENT_ADP	Address of the ADP in the preceding level of the system's ADP hierarchy that is related to this ADP and its peers. In the system ADP, this field contains a zero.
ADP$PS_PEER_ADP	Address of the next ADP in the list of ADPs that are children of a common parent ADP in the preceding level of the system's ADP hierarchy, and headed by field ADP$PS_CHILD_ADP in that parent ADP. This field contains a zero if the ADP has no peers.

(continued on next page)

Table 10–1 (Cont.) Contents of Adapter Control Block

Field	Use	
ADP$PS_CHILD_ADP	Listhead of the ADPs that are related to this ADP in the succeeding level of the ADP hierarchy, or zero if the ADP has no children. At this lower level, the child ADPs of a common parent ADP are linked together by the contents of their ADP$PS_PEER_ADP fields.	
ADP$L_PROBE_CMD	Index into the adapter command table that EXE$TEST_CSR uses to determine which command to use when probing the interconnect described by this ADP.	
ADP$PS_BUS_ARRAY	Address of BUSARRAY describing the nodes on the tightly coupled interconnect or the ports of a multichannel I/O widget or controller associated with this ADP.	
ADP$PS_COMMAND_TBL	Address of the adapter command table specific to the I/O interconnect described by this ADP. The OpenVMS adapter initialization routine constructs this table. IOC$CRAM_CMD refers to this field to locate the table when it calculates the COMMAND, MASK, and RBADR fields of a hardware I/O mailbox involved in a transaction to a device interface register.	
ADP$PS_SPINLOCK	Address of device lock synchronizing access to the CSRs of the devices associated with this ADP. The OpenVMS adapter initialization routine allocates this device lock and places its address in this field, IDB$PS_SPL, and CRB$PS_DLCK.	
ADP$W_PRIM_NODE_NUM	Node number of the I/O adapter (or widget) on the local interconnect (for instance, the node number of the DEC 7000 Alpha Model 600 system bus [PMI] to XMI bus adapter on the PMI).	
ADP$W_SEC_NODE_NUM	Node number of the I/O adapter on the remote interconnect (for instance, the node number of the DEC 7000 Alpha Model 600 system bus [PMI] to XMI bus adapter on the XMI).	
ADP$B_HOSE_NUM	Hose number associated with the I/O adapter. OpenVMS adapter initialization routine writes this field.	
ADP$L_CRAB	Address of CRAB used to manage map registers, if the Alpha system provides map registers for this adapter.	
ADP$L_ADAPTER_FLAGS	The following bit is defined within ADP$L_ADAPTER_FLAGS:	
	ADP$V_INDIRECT_VECTOR	Adapter services indirectly vectored interrupts for its associated devices.
	ADP$V_ONLINE	Adapter is online.
	ADP$V_BOOT_ADP	Adapter is boot adapter.

(continued on next page)

Table 10–1 (Cont.) Contents of Adapter Control Block

Field	Use	
ADP$L_VPORTSTS	CI-VAX port status bits. The following bits are defined within ADP$L_VPORTSTS:	
	ADP$V_SHUTDOWN	CI-adapter microcode is stopped.
	ADP$V_PORTONLY	CI-port restart only—no adapter restart.
	ADP$V_STRUCT_ALLOCATED	CI/SCSI–adapter-wide structures allocated.
ADP$PS_NODE_FUNCTION	Procedure value of the node-specific function routine that services driver calls to IOC$NODE_FUNCTION.	
ADP$L_AVECTOR	Address of first SCB vector for adapter.	
ADP$Q_SCRATCH_BUF_PA	Physical address of adapter scratch buffer.	
ADP$PS_SCRATCH_BUF_VA	Virtual address of a physically contiguous scratch buffer used in an adapter-specific manner.	
ADP$L_SCRATCH_BUF_LEN	Size of adapter scratch buffer.	
ADP$L_LSDUMP	Address of physical contiguous memory for the adapter memory dump.	
ADP$PS_PROBE_CSR	Procedure value of adapter-specific routine that checks for the existence of devices on an I/O interconnect. EXE$PROBE_CSR issues a standard call to this routine.	
ADP$PS_PROBE_CSR_CLEANUP	Procedure value of adapter probe CSR cleanup routine. The adapter-specific probe CSR routine calls the cleanup routine when an error occurs during its attempts to probe an I/O interconnect.	
ADP$PS_LOAD_MAP_REG	Procedure value of adapter load map register routine.	
ADP$PS_SHUTDOWN	Procedure value of adapter shutdown routine.	
ADP$PS_CONFIG_TABLE	Pointer to autoconfiguration table.	
ADP$PS_MAP_REG_BASE	Base virtual address of adapter map registers.	
ADP$PS_ADP_SPECIFIC	Address of adapter auxiliary data structure.	
ADP$PS_DISABLE_INTERRUPTS	Address of adapter-specific interrupt disabling routine.	
ADP$PS_STARTUP	Address of adapter-specific startup routine.	
ADP$PS_INIT	Address of adapter-specific initialization routine.	
ADP$Q_HARDWARE_TYPE	Saved hardware device type information. The interpretation of this information is adapter-specific.	
ADP$Q_HARDWARE_REV	Saved hardware device revision information. The interpretation of this information is adapter-specific.	

(continued on next page)

Table 10–1 (Cont.) Contents of Adapter Control Block

Field	Use
ADP$L_INTD	Interrupt transfer vector. For adapters that service indirect interrupts (ADP$V_INDIRECT_VECTOR in ADP$L_ADAPTER_FLAGS is set), this 4-longword field (described in Section 10.5) provides information used by OpenVMS Alpha to service a device interrupt, such as the location of the ADP and its indirect interrupt service routine. See the description of the ADP$L_VECTOR field for additional information on how the adapter services indirect interrupts.

10.1.1 BUSARRAY (Bus Array)

The bus array (BUSARRAY) contains information about the nodes on a tightly coupled I/O interconnect or the ports of a multichannel I/O widget. The BUSARRAY consists of a fixed portion and an array of entries. The fixed portion records the interconnect type, the number of nodes on the interconnect, and a pointer to the ADP with which the BUSARRAY is associated. Each array entry records the node number, the node's hardware ID, and a pointer to either an ADP or a CRB.

Table 10–2 describes the fields of the BUSARRAY structure; Table 10–3 describes the contents of each entry in the bus array.

Table 10–2 Contents of Bus Array

Field	Use	
BUSARRAY$PS_PARENT_ADP	Address of ADP for the tightly coupled I/O interconnect or multichannel I/O widget the BUSARRAY describes.	
BUSARRAY$W_SIZE	Size of BUSARRAY in bytes. The adapter initialization routine writes this field when it creates the BUSARRAY.	
BUSARRAY$B_TYPE	Type of data structure. The adapter initialization routine writes the symbolic constant DYN$C_MISC in this field when it creates the BUSARRAY.	
BUSARRAY$B_SUBTYPE	Structure subtype. The adapter initialization routine writes DYN$C_BUSARRAY in this field when it creates the BUSARRAY.	
BUSARRAY$L_BUS_TYPE	Type of tightly coupled I/O interconnect or multichannel I/O widget the BUSARRAY describes. The adapter initialization routine writes this field when it creates the BUSARRAY. The following constants (defined by the $BUSDEF macro in SYS$LIBRARY:LIB.MLB) represent the interconnects supported on OpenVMS Alpha systems:	
	BUS$_FBUS	Futurebus
	BUS$_XMI	XMI
	BUS$_LBUS	DEC 4000 Alpha LBUS
	BUS$_TURBO	TURBOchannel

(continued on next page)

Table 10–2 (Cont.) Contents of Bus Array

Field	Use	
	BUS$_CBUS	DEC 4000 Alpha system bus
	BUS$_LSB	DEC 7000 Alpha Model 600 system bus
	BUS$_SCSI	SCSI
	BUS$_NI	Ethernet
	BUS$_CI	CI
	BUS$_KA0402_CORE_IO	DEC 3000 Alpha Model 500 core I/O bus
	BUS$_KDM70	KDM70
	BUS$_GENXMI	Generic XMI
	BUS$_BUSLESS_SYSTEM	No bus
BUSARRAY$L_BUS_NODE_CNT	Number of entries in the bus array located at BUSARRAY$Q_ENTRY_LIST. The OpenVMS adapter initialization routine writes this field when it creates the BUSARRAY.	
BUSARRAY$Q_ENTRY_LIST	Bus array consisting of BUSARRAY$L_BUS_NODE_CNT entries.	

Table 10–3 Contents of Bus Array

Field	Use
BUSARRAY$Q_HW_ID	Hardware ID. The macro $NDTDEF (in SYS$LIBRARY:LIB.MLB) provides symbolic definitions for the hardware IDs of all possible OpenVMS Alpha nodes.
BUSARRAY$Q_CSR	Base address of the node's CSR. The adapter initialization routine writes this field.
BUSARRAY$L_NODE_NUMBER	Node number. The adapter initialization routine writes this field.
BUSARRAY$L_FLAGS	Bus array flags. The only bit that is currently defined, BUSARRAY$V_NO_RECONNECT, when set, indicates that a node has been configured properly. A bus-specific routine in an IOGEN configuration building module (ICBM) sets this bit.
BUSARRAY$PS_CRB	Pointer to node's CRB. This field must be zero if BUSARRAY$PS_ADP is filled in.
BUSARRAY$PS_ADP	Pointer to the child ADP of the parent ADP (identified by BUSARRAY$PS_PARENT_ADP) with which this node is associated. If there is no such child ADP, this field must be zero.
BUSARRAY$L_AUTOCONFIG	Reserved for the Autoconfiguration facility.
BUSARRAY$L_CTRLLTR	A bus-specific routine in an IOGEN configuration building modules writes this field by calling IOGEN$ASSIGN_CONTROLLER.

10.2 CCB (Channel Control Block)

When a process assigns an I/O channel to a device unit with the $ASSIGN system service, EXE$ASSIGN locates a free block among the channel control blocks (CCBs) preallocated to the process. EXE$ASSIGN then writes into the CCB a description of the device attached to the CCB's channel.

The channel control block is the only data structure described in this chapter that exists in the control (P1) region of a process address space. It is described in Table 10–4.

Table 10–4 Contents of Channel Control Block

Field	Use	
CCB$L_UCB	Address of UCB of assigned device unit. EXE$ASSIGN writes a value into this field. EXE$QIO reads this field to determine that the I/O request specifies a process I/O channel assigned to a device and to obtain the device's UCB address.	
CCB$L_WIND	Address of window control block (WCB) for file-structured device assignment. This field is written by an ancillary control process (ACP) or the extended QIO processor (XQP) and read by EXE$QIO. A file-structured device's XQP or ACP creates a WCB when a process accesses a file on a device assigned to a process I/O channel. The WCB maps the virtual block numbers of the file to a series of physical locations on the device.	
CCB$L_STS	Channel status. The following bits are defined within CCB$L_STS:	
	CCB$V_AMB	Mailbox associated with channel.
	CCB$V_IMGTMP	Temporary image.
	CCB$V_RDCHKDON	Read protection check completed.
	CCB$V_WRTCHKDON	Write protection check completed.
	CCB$V_LOGCHKDON	Logical I/O access check done.
	CCB$V_PHYCHKDON	Physical I/O access check done.
CCB$B_AMOD	Access mode plus 1 of the channel. EXE$ASSIGN writes the access mode value into this field.	
CCB$L_IOC	Number of outstanding I/O requests on channel. EXE$QIO increases this field when it begins to process an I/O request that specifies the channel. During I/O postprocessing, the special kernel-mode AST routine decrements this field. Some FDT routines and EXE$DASSGN read this field.	

(continued on next page)

Table 10–4 (Cont.) Contents of Channel Control Block

Field	Use
CCB$L_DIRP	Address of I/O request packet (IRP) for requested deaccess. A number of outstanding I/O requests can be pending on the same process I/O channel at one time. If the process that owns the channel issues an I/O request to deaccess the device, EXE$QIO holds the deaccess request until all other outstanding I/O requests are processed.
CCB$L_CHAN	Associated channel number.

10.3 CRAM (Controller Register Access Mailbox)

The controller register access mailbox (CRAM) contains information that describes a specific hardware I/O mailbox transaction. To facilitate mailbox operations within the operating system, the CRAM contains information required by the operating system as well as the hardware I/O mailbox itself. For example, mailbox operations require the physical address of the hardware mailbox itself as well as the virtual address of the corresponding mailbox pointer register (MBPR). Additionally, the timeout values for both the queuing and waiting portions of a mailbox operation are kept in the CRAM.

CRAMs are allocated from pages obtained from the memory management free list. Once the pages have been allocated from the free list, they are managed privately by the CRAM allocation and deallocation code. Each page of CRAMs begins with a structure known as a controller register access mailbox header (CRAMH); the set of pages is maintained as a linked list starting at IOC$GQ_CRAMH_HDR.

The controller register access mailbox is described in Table 10–5.

Table 10–5 Contents of Controller Register Access Mailbox

Field	Use
CRAM$L_FLINK	Forward link to next CRAM in list (headed by IDB$PS_CRAM or UCB$PS_CRAM). The driver-loading procedure initializes this field when the driver preallocates CRAMs by specifying the **idb_crams** or **ucb_crams** argument to the DPTAB macro. The contents of this field are unpredictable and must be managed by the driver when it spontaneously allocates CRAMs.
CRAM$L_BLINK	Backward link to next CRAM in list (headed by IDB$PS_CRAM or UCB$PS_CRAM). The driver-loading procedure initializes this field when the driver preallocates CRAMs by specifying the **idb_crams** or **ucb_crams** argument to the DPTAB macro. The contents of this field are unpredictable and must be managed by the driver when it spontaneously allocates CRAMs.
CRAM$W_SIZE	Size of CRAM in bytes. IOC$ALLOCATE_CRAM writes the symbolic constant CRAM$K_LENGTH in this field when it initializes the CRAM.
CRAM$B_TYPE	Structure type. IOC$ALLOCATE_CRAM initializes this field to DYN$C_MISC.

(continued on next page)

Table 10–5 (Cont.) Contents of Controller Register Access Mailbox

Field	Use
CRAM$B_SUBTYPE	Structure subtype. IOC$ALLOCATE_CRAM initializes this field to DYN$C_CRAM.
CRAM$L_MBPR	Virtual address of mailbox pointer register (MBPR). When IOC$ALLOCATE_CRAM is called by the driver-loading procedure, or when it is called independently with the **idb** parameter, it initializes this field from the contents of ADP$PS_MBPR. Otherwise, it places a zero in this field.
CRAM$Q_HW_MBX	Physical address of hardware mailbox. IOC$ALLOCATE_CRAM initializes this field.
CRAM$Q_QUEUE_TIME	MBPR queue timeout interval in nanoseconds. If IOC$CRAM_QUEUE or IOC$CRAM_CMD cannot queue the hardware I/O mailbox defined in this CRAM to the MBPR in this amount of time, it returns SS$_INTERLOCK status to its caller. When IOC$ALLOCATE_CRAM is called by the driver-loading procedure, or when it is called independently with the **idb** parameter, it initializes this field from the contents of ADP$Q_QUEUE_TIME. Otherwise, it places a zero in this field.
CRAM$Q_WAIT_TIME	Mailbox transaction wait timeout interval in nanoseconds. If IOC$CRAM_IO or IOC$CRAM_WAIT does not see the done or error bit set in the hardware mailbox in this interval, it returns SS$_TIMEOUT status to its caller. When IOC$ALLOCATE_CRAM is called by the driver-loading procedure, or when it is called independently with the **idb** parameter, it initializes this field from the contents of ADP$Q_WAIT_TIME. Otherwise, it places a zero in this field.
CRAM$L_DRIVER	Spare longword for driver use.
CRAM$L_IDB	Pointer to IDB. IOC$ALLOCATE_CRAM initializes this field when called from the driver-loading procedure, and when called with a nonzero **idb** parameter. Otherwise, it places a zero in this field.
CRAM$L_UCB	Pointer to UCB. IOC$ALLOCATE_CRAM initializes this field when called from the driver-loading procedure (if the **ucb_cram** argument is supplied to the DPTAB macro), and when called with a nonzero **ucb** parameter. Otherwise, it places a zero in this field.

(continued on next page)

Table 10–5 (Cont.) Contents of Controller Register Access Mailbox

Field	Use	
CRAM$L_CRAM_FLAGS	The following bits are defined within CRAM$L_CRAM_FLAGS:	
	CRAM$V_CRAM_IN_USE	CRAM is valid. IOC$CRAM_QUEUE and IOC$CRAM_IO set this bit when they have successfully posted the hardware I/O mailbox portion of the CRAM to the MBPR. IOC$CRAM_IO and IOC$CRAM_WAIT clear this bit when the mailbox transaction is completed (either successfully or unsuccessfully) within the mailbox transaction timeout interval (CRAM$Q_WAIT_TIME).
	CRAM$V_DER	Disable error reporting.
CRAM$L_COMMAND	Command to the remote I/O interconnect command specifying a read or write transaction. The local I/O adapter delivers this command to the remote interconnect to which the target widget is connected. The command may also include fields such as address only, address width, and data width. This field, aligned on a 64-byte boundary, indicates the beginning of the hardware I/O mailbox structure in this CRAM. The characters "MBZ" (must be zero) indicate that the field must contain a zero when it is supplied in a CRAM operation. Given a command index, IOC$CRAM_CMD initializes this field in a manner specific to the I/O interconnect that is to be the target of an operation using this CRAM.	
CRAM$B_BYTE_MASK	Byte mask that indicates which bytes within the remote bus address (CRAM$Q_RBADR) are to be written for mailbox write operations. IOC$CRAM_CMD, on behalf of a device driver, writes the size of the target location (byte, word, longword, or quadword) in this field. Given a byte offset to an address in remote I/O space, IOC$CRAM_CMD initializes this field in a manner specific to the masking mode of the I/O interconnect that is to be the target of an operation using this CRAM.	
CRAM$B_HOSE	I/O bus number, or hose. This field specifies the remote I/O interconnect to be accessed by the mailbox transaction described by this CRAM. When IOC$ALLOCATE_CRAM is called by the driver-loading procedure, or when it is called independently with the **idb** parameter, it initializes this field from the contents of ADP$B_HOSE_NUM. Otherwise, it places a zero in this field.	

(continued on next page)

Table 10–5 (Cont.) Contents of Controller Register Access Mailbox

Field	Use	
CRAM$Q_RBADR	Remote bus address. A device driver calls IOC$CRAM_CMD to write a value in this field that represents the physical address of the device interface register to be accessed. IOC$CRAM_CMD calculates this value from IDB$Q_CSR (or ADP$Q_CSR if IDB$Q_CSR is not available) and the **byte_offset** input argument.	
CRAM$Q_WDATA	Data to be written. If CRAM$L_COMMAND indicates a write transaction to the remote interconnect, the driver initializes this field with the data to be written to the target device interface register. If CRAM$L_COMMAND indicates a read transaction, this field is not used.	
CRAM$Q_RDATA	Returned read data. If CRAM$L_COMMAND indicates a read transaction to the remote interconnect, the remote adapter returns the requested data in this field. If CRAM$L_COMMAND indicates a write transaction, the contents of this field are unpredictable.	
CRAM$W_MBX_FLAGS	The following bits are defined within CRAM$W_MBX_FLAGS:	
	CRAM$V_MBX_DONE	Mailbox operation completed. IOC$CRAM_WAIT and IOC$CRAM_IO check this bit to determine the completion of a hardware I/O mailbox transaction. For both read and write commands, this bit, when set, indicates that the CRAM$V_MBX_ERROR, CRAM$W_ERROR_BITS, and CRAM$Q_RDATA fields are valid. The mailbox structure may then be safely modified by software (reused). Note that the setting of the DON bit does not guarantee that a remote I/O space write has actually completed at the bridge.
	CRAM$V_MBX_ERROR	Error in operation. IOC$CRAM_WAIT and IOC$CRAM_IO check this bit to determine whether an error occurred during a hardware I/O mailbox transaction. If set on a read command, indicates that an error was encountered and that the CRAM$W_ERROR_BITS field contains additional information. This bit is valid only when CRAM$V_MBX_DONE is set.
CRAM$W_ERROR_BITS	Device-specific error bits that indicate the completion status of a mailbox transaction described by this CRAM.	

10.4 CRB (Channel Request Block)

The activity of each controller in a configuration is described in a channel request block (CRB). This data structure contains pointers to the wait queue of driver fork processes waiting to gain access to a device through the controller. It also contains one interrupt transfer vector (VEC) for each of the controller's interrupt vectors.

The channel request block is described in Table 10–6.

Table 10–6 Contents of Channel Request Block

Field	Use
CRB$L_FQFL	Fork queue forward link. The link points to the next entry in the fork queue. Controller initialization routines write this field when they must drop IPL to utilize certain executive routines, such as those that allocate CRAMs or nonpaged memory, that must be called at a lower IPL. The CRB timeout mechanism also uses the CRB fork block to lower IPL prior to calling the CRB timeout routine.
CRB$L_FQBL	Fork queue backward link. The link points to the previous entry in the fork queue.
CRB$W_SIZE	Size of CRB in bytes. The driver-loading procedure writes this field when it creates the CRB.
CRB$B_TYPE	Type of data structure. The driver-loading procedure writes the symbolic constant DYN$C_CRB into this field when it creates the CRB.
CRB$B_FLCK	Fork lock at which the controller's fork operations are synchronized. If it must use the CRB fork block, a driver either uses a DPT_STORE macro to initialize this field or explicitly sets its value within the controller initialization routine.
CRB$L_FPC	Procedure value of routine at which execution resumes when the fork dispatcher dequeues the fork block. EXE$PRIMITIVE_FORK writes this field when called to suspend driver execution.
CRB$Q_FR3	Value of R3 at the time that the executing code requests the operating system to create a fork block. EXE$PRIMITIVE_FORK writes this field when called to suspend driver execution.
CRB$Q_FR4	Value of R4 at the time that the executing code requests OpenVMS to create a fork block. EXE$PRIMITIVE_FORK writes this field when called to suspend driver execution.
CRB$B_TT_TYPE	Controller type.
CRB$L_REFC	Unit control block (UCB) reference count. The driver-loading procedure increases the value in this field each time it creates a UCB for a device attached to the controller.

(continued on next page)

Table 10–6 (Cont.) Contents of Channel Request Block

Field	Use	
CRB$B_MASK	Mask that describes controller status. The following fields are defined in CRB$B_MASK:	
	CRB$V_BSY	Busy bit. IOC$PRIMITIVE_REQCHAN*y* reads the busy bit to determine whether the controller is free and sets this bit when it allocates the controller data channel to a driver. IOC$RELCHAN clears the busy bit if no driver is waiting to acquire the channel.
	CRB$V_UNINIT	Indication, when set, that the OpenVMS driver loading procedure has yet to call the driver's controller initialization routine. The driver loading procedure reads this bit to determine whether to call the controller initialization routine and clears it when the initialization routine completes.
CRB$PS_BUSARRAY	Address of BUSARRAY that describes the devices residing on loosely coupled I/O interconnects (for instance, a SCSI port).	
CRB$Q_AUXSTRUC	Address of auxiliary data structure used by device driver to store special controller information. A device driver requiring such a structure generally allocates a block of nonpaged dynamic memory in its controller initialization routine and places a pointer to it in this field.	
CRB$Q_LAN_STRUC	Address of auxiliary data structure used by local area network drivers.	
CRB$Q_SSB_STRUC	Address of auxiliary data structure used by system communications services drivers.	
CRB$L_TIMELINK	Forward link in queue of CRBs waiting for periodic wakeups. This field points to the CRB$L_TIMELINK field of the next CRB in the list. The CRB$L_TIMELINK field of the last CRB in the list contains zero. The listhead for this queue is IOC$GL_CRBTMOUT. Use of this field is reserved to Digital.	
CRB$L_NODE	Bus-slot number of the controller node. The OpenVMS Alpha driver-loading procedure initializes this field, which is used by IOC$NODE_FUNCTION to enable or disable functionality for the node.	
CRB$L_DUETIME	Time in seconds, relative to EXE$GL_ABSTIM, at which next periodic wakeup associated with the CRB is to be delivered. Compute this value by raising IPL to IPL$_POWER, adding the required number of seconds to the contents of EXE$GL_ABSTIM, and storing the result in this field. Use of this field is reserved to Digital.	

(continued on next page)

Table 10–6 (Cont.) Contents of Channel Request Block

Field	Use
CRB$L_TOUTROUT	Procedure value of routine to be called at fork IPL (holding a corresponding fork lock if necessary) when a periodic wakeup associated with CRB becomes due. The routine must compute and reset the value in CRB$L_DUETIME if another periodic wakeup request is desired. Use of this field is reserved to Digital.
CRB$PS_DLCK	Address of controller's device lock. The driver-loading procedure initializes this field and propagates it to each UCB it creates for the device units associated with the controller.
CRB$PS_CRB_LINK	Pointer to next CRB on ADP.
CRB$PS_CTRLR_SHUTDOWN	Procedure value of driver controller shutdown routine.
CRB$L_INTD	Interrupt transfer vector. This 4-longword field (described in Section 10.5) contains information used by the operating system to service a device interrupt, such as the location of the device's interrupt service routine and its associated interrupt dispatch block (IDB).
CRB$L_INTD2	Second interrupt transfer vector for devices with multiple interrupt vectors.

10.5 VEC (Interrupt Transfer Vector Block)

An interrupt transfer vector block (VEC) exists in OpenVMS only as a substructure of a CRB or an ADP. A VEC stores information that allows OpenVMS to correctly dispatch and service the interrupts of devices that share a common controller or adapter. The VEC substructures of ADPs are of interest only to OpenVMS-supplied device drivers.

By default, the driver-loading procedure creates a single VEC within a given CRB. (Adapter initialization code generates the VECs associated with an ADP.) You can control the number of VECs created by specifying a value in the /NUMVEC qualifier of an SYSMAN IO CONNECT command.

The OpenVMS driver-loading procedure initializes the contents of each VEC's IDB and ADP pointers and connects the VEC to the appropriate vector offsets within the system control block (SCB). A device driver must initialize the VEC$PS_ISR_CODE and VEC$PS_ISR_PD fields in each VEC by invoking the DPT_STORE_ISR macro, as described in Chapter 11.

Although the OpenVMS interrupt dispatching mechanism passes the address of the device's IDB to a driver's interrupt service routine as its sole parameter, other driver routines must determine the location of the IDB by directly accessing VEC$L_IDB in a VEC substructure. The data structure definition macro $CRBDEF supplies symbolic offsets so that a driver can easily locate the first two VECs. For additional VECs, the driver must employ the following formula, where *n* represents the vector number:

CRB$L_INTD+((*n*-1)*VEC$K_LENGTH)

The following table lists the symbolic location of the first three VECs for a given controller:

Vector Number	Symbolic Offset to VEC
1	CRB$L_INTD
2	CRB$L_INTD2
3	CRB$L_INTD+<2*VEC$K_LENGTH>

Table 10–7 describes the contents of the VEC substructure.

Table 10–7 Contents of Interrupt Transfer Vector Block (VEC)

Field	Use
VEC$PS_ISR_CODE	Address of the code entry point of a driver interrupt service routine (ISR). The driver specifies an ISR by using the DPT_STORE_ISR macro, which initializes this field.
VEC$PS_ISR_PD	Address of the procedure descriptor of a driver ISR. The driver specifies an ISR by using the DPT_STORE_ISR macro, which initializes this field.
VEC$L_IDB	Address of IDB for controller. The driver-loading procedure creates an IDB for each CRB and loads the address of the IDB in this field. Device drivers use the IDB address to obtain the addresses of IDB CRAMs. When a driver's interrupt service routine gains control, it receives this value as its only parameter.
VEC$PS_ADP	Address of ADP. The SYSMAN command IO CONNECT must specify the nexus number of the adapter used by a controller. The driver-loading procedure writes the address of the ADP for the specified adapter into the VEC$PS_ADP field.

10.6 DDB (Device Data Block)

The device data block (DDB) is a block that identifies the generic device/controller name and driver name for a set of devices attached to a single controller. The driver-loading procedure creates a DDB for each controller during autoconfiguration at system startup and dynamically creates additional DDBs for new controllers as they are added to the system using the SYSMAN command CONNECT. The procedure initializes all fields in the DDB. All the DDBs associated with a given system block (SB) are linked in a singly linked list off that SB. The field DDB$L_SB points to the parent SB of any given DDB.

The device data block is described in Table 10–8.

Table 10–8 Contents of Device Data Block

Field	Use
DDB$L_LINK	Address of next DDB. A zero indicates that this is the last DDB in the DDB chain.
DDB$L_UCB	Address of UCB for first unit attached to controller.

(continued on next page)

Table 10–8 (Cont.) Contents of Device Data Block

Field	Use
DDB$W_SIZE	Size of DDB in bytes. The driver-loading procedure writes the symbolic constant DDB$K_LENGTH in this field when it creates the DDB.
DDB$B_TYPE	Type of data structure. The driver-loading procedure writes the constant DYN$C_DDB into this field when the procedure creates the DDB.
DDB$L_DDT	Address of driver dispatch table (DDT). OpenVMS can transfer control to a device driver only through procedure values and entry points listed in the DDT, CRB, and UCB fork block. The driver-loading procedure initializes this field.
DDB$L_ACPD	Name of default ACP (or XQP) for controller. ACPs that control access to file-structured devices (or the XQP) use the high-order byte of this field, DDB$B_ACPCLASS, to indicate the class of the file-structured device. If the ACP_MULTIPLE system parameter is set, the initialization procedure creates a unique ACP for each class of file-structured device. Drivers initialize DDB$B_ACPCLASS by invoking a DPT_STORE macro. Values for DDB$B_ACPCLASS are as follows: DDB$K_PACK Standard disk pack DDB$K_CART Cartridge disk pack DDB$K_SLOW Floppy disk DDB$K_TAPE Magnetic tape that simulates file-structured device
DDB$T_NAME	Name of device. The first byte of this field contains the number of characters in the device name. The remainder of the field contains a string of up to 15 characters representing the device name in the format *ddc*, where dd = device code (up to 9 alphabetic characters) c = controller designation (alphabetic)
DDB$PS_DPT	Address of DPT of driver that supports this device.
DDB$PS_DRVLINK	Address of next DDB in singly linked list, headed by DPT$PS_DDB_LIST, of DDBs serviced by a particular driver.
DDB$L_SB	Address of system block.
DDB$L_CONLINK	Address of next DDB in the connection subchain.
DDB$L_ALLOCLS	Allocation class of device.
DDB$L_2P_UCB	Address of the first UCB on the secondary path.

10.7 DDT (Driver Dispatch Table)

Each device driver contains a driver dispatch table (DDT). The DDT lists procedure values for driver entry points that system routines call.

A device driver creates a DDT by invoking the VAX MACRO DDTAB macro. Table 10–9 describes the fields in the driver dispatch table.

Table 10–9 Contents of Driver Dispatch Table

Field	Use
DDT$PS_START_2	Procedure value of the driver's start-I/O routine. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **start** argument to the macro. All drivers must specify a start-I/O routine. When a device unit is idle and an I/O request is pending for that unit, IOC$INITIATE transfers control to the routine entry point represented by the procedure value in this field. A driver that employs kernel process services typically specifies its start-I/O routine in the **kp_startio** argument to the DDTAB macro, and the system routine EXE$KP_STARTIO in the **start** argument. This allows OpenVMS to set up the kernel process environment prior to transferring control to the driver's start-I/O routine.
DDT$PS_START_JSB	Procedure value of the driver Start I/O routine when DDTAB JSB_START is used. The DDT$PS_START field contains a pointer to the IOC$START_C2J routine.
DDT$IW_SIZE	Size of DDT in bytes. The DDTAB macro writes the symbolic constant DDT$K_LENGTH in this field when creating the DDT.
DDT$W_DIAGBUF	Size of diagnostic buffer, as specified in the **diagbf** argument to the DDTAB macro. The value is the size in bytes of a diagnostic buffer for the device. When EXE$QIO preprocesses an I/O request, it allocates a system buffer of the size recorded in this field (if it contains a nonzero value) if the process requesting the I/O has DIAGNOSE privilege and specifies a diagnostic buffer in the I/O request. IOC$DIAGBUFILL fills the buffer after the I/O operation completes.
DDT$W_ERRORBUF	Size of error message buffer, as specified in the **erlgbf** argument to the DDTAB macro. The value is the size in bytes of an error message buffer for the device. If error logging is enabled and an error occurs during an I/O operation, the driver calls ERL$DEVICERR or ERL$DEVICTMO to allocate and write error-logging data into the error message buffer. IOC$INITIATE and IOC$REQCOM write values into the buffer if an error has occurred.
DDT$W_FDTSIZE	Unused on OpenVMS Alpha systems.
DDT$PS_CTRLINIT_2	Procedure value of controller initialization routine. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **ctrlinit** argument to the macro.
DDT$PS_UNITINIT_2	Procedure value of the device's unit initialization routine. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **unitinit** argument to the macro.

(continued on next page)

Table 10–9 (Cont.) Contents of Driver Dispatch Table

Field	Use
DDT$PS_CLONEDUCB_2	Procedure value of cloned UCB routine. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **cloneducb** argument to the macro.
DDT$PS_FDT_2	Address of the driver's FDT. Every driver must specify this address in the **functb** argument to the DDTAB macro. EXE$QIO refers to the FDT to validate I/O function codes, decide which functions are buffered, and call FDT routines associated with function codes.
DDT$PS_CANCEL_2	Procedure value of the driver's cancel-I/O routine. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **cancel** argument to the macro. Some devices require special cleanup processing when a process or a system routine cancels an I/O request before the I/O operation completes or when the last channel is deassigned. The $DASSGN, $DALLOC, and $CANCEL system services cancel I/O requests.
DDT$PS_REGDUMP_2	Procedure value of the driver's register dumping routine. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **regdmp** argument to the macro. IOC$DIAGBUFILL, ERL$DEVICERR, and ERL$DEVICTMO call this routine to write device register contents into a diagnostic buffer or error message buffer.
DDT$PS_ALTSTART_2	Procedure value of the driver's alternate start-I/O routine. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **altstart** argument to the macro. EXE$ALTQUEPKT transfers control to the alternate start-I/O routine specified in this field.
DDT$PS_ALTSTART_JSB	Procedure value of the driver Alternate Start I/O routine when DDTAB JSB_ALTSTART is used. The DDT$PS_ALTSTART field contains a pointer to the IOC$ALTSTART_C2J routine.
DDT$PS_MNTVER_2	Procedure value of the system routine (IOC$MNTVER) called at the beginning and end of mount verification operation. The default value of the **mntver** argument to the DPTAB macro is the procedure value of this routine. Use of the **mntver** argument to specify any routine other than IOC$MNTVER is reserved to Digital.
DDT$L_MNTV_SSSC	Procedure value of the routine that is called when mount verification is performed for a shadow-set state change. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **mntv_sssc** argument to the macro. Use of this field is reserved to Digital.

(continued on next page)

Table 10–9 (Cont.) Contents of Driver Dispatch Table

Field	Use
DDT$L_MNTV_FOR	Procedure value of the routine that is called when mount verification is performed for a foreign device. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **mntv_for** argument to the macro. Use of this field is reserved to Digital.
DDT$L_MNTV_SQD	Procedure value of the routine that is called when mount verification is performed for a sequential device. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **mntv_sqd** argument to the macro. Use of this field is reserved to Digital.
DDT$L_AUX_STORAGE	Address of auxiliary storage area, as specified in the **aux_storage** argument to the DDTAB macro. Use of this field is reserved to Digital.
DDT$L_AUX_ROUTINE	Procedure value of auxiliary routine in the mailbox driver that is called by SYS$ASSIGN. The OpenVMS VAX mailbox driver uses this routine to complete the processing of reader-wait and writer-wait set mode requests. (Auxiliary routines have yet to be implemented in OpenVMS Alpha systems.) The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **aux_routine** argument to the macro. Use of this field is reserved to Digital.
DDT$PS_CHANNEL_ASSIGN_2	Procedure value of routine, called by SYS$ASSIGN, to complete channel assignment in a device-specific manner. (Channel-assignment routines have yet to be implemented in OpenVMS Alpha systems.) The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **channel_assign** argument to the macro. Use of this field is reserved to Digital.
DDT$PS_CANCEL_SELECTIVE_2	Procedure value of the routine that cancels a list of I/O requests from the specified channel, including both waiting and active requests. The OpenVMS VAX terminal driver and mailbox driver provide this capability which is not yet implemented in OpenVMS Alpha systems. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **cancel_selective** argument to the macro. Use of this field is reserved to Digital.
DDT$IS_STACK_BCNT	Size in bytes of the kernel process stack, as indicated by the **kp_stack_size** argument to the DDTAB macro. EXE$KP_STARTIO uses this value, or KPB$K_MIN_IO_STACK (currently 8KB), whichever is larger, to determine the size of the stack created for the driver's start I/O kernel process thread.

(continued on next page)

Table 10–9 (Cont.) Contents of Driver Dispatch Table

Field	Use
DDT$IS_REG_MASK	Kernel process register save mask, as indicated by the **kp_reg_mask** argument to the DDTAB macro. Each time a kernel process is stalled and restarted, any registers that the thread uses other than registers that the calling standard defines as scratch must be saved. EXE$KP_STARTIO establishes this set of registers by merging the mask specified in this field with a register save mask (represented by the symbolic constant KPREG$K_MIN_IO_REG_MASK) that includes R2 through R5, R12 through R15, R26, R27, and R29. It then specifies the resulting mask in its call to EXE$KP_START. It is this latter mask that EXE$KP_START stores in KPB$IS_REG_MASK for the lifetime of the kernel process. Note that R0, R1, R16 through R25, R28, R30, and R31 are never preserved and are illegal in a register save mask. OpenVMS represents the set of these registers by the symbolic constant KPREG$K_ERR_REG_MASK. If any of these registers are indicated by the contents of DDTIS_REG_MASK, EXEKP_START removes them from the mask it stores in the KPB.
DDT$PS_KP_STARTIO	Procedure value of the start-I/O routine of a driver that employs the kernel process services. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **kp_startio** argument to the macro. Such a driver typically specifies the system routine EXE$KP_STARTIO in the **start** argument to the DDTAB macro. EXE$KP_STARTIO calls the start-I/O routine specified in this field after setting up the kernel process environment.

10.8 DPT (Driver Prologue Table)

When loading a device driver and its database into virtual memory, the driver-loading procedure finds the basic description of the driver and its device in a driver prologue table (DPT). The DPT provides the length, name, adapter type, and loading and reloading specifications for the driver.

A device driver creates a DPT by invoking the DPTAB macro. Table 10–10 describes the driver prologue table.

Table 10–10 Contents of Driver Prologue Table

Field	Use
DPT$L_FLINK	Forward link to next DPT. The driver-loading procedure writes this field. The procedure links all DPTs in the system in a doubly linked list.
DPT$L_BLINK	Backward link to previous DPT. The driver-loading procedure writes this field.

(continued on next page)

Table 10–10 (Cont.) Contents of Driver Prologue Table

Field	Use
DPT$W_SIZE	Size of DPT in bytes. The DPTAB macro writes the value *DPT$K_BASE_LEN + NAM$C_MAXRSS* in this field when it creates the DPT.
DPT$B_TYPE	Type of data structure. The DPTAB macro always writes the symbolic constant DYN$C_DPT into this field.
DPT$IW_STEP	OpenVMS Alpha driver step number. You must indicate that a given driver conforms to the coding practices for a Step 2 driver by supplying **step**=**2** in the DPTAB macro invocation. Consequently, the DPTAB macro writes the symbol constant DPT$K_STEP_2 in this field.
DPT$IW_STEPVER	Integer signifying the version of Step 2 interface used by this driver. An increment of this value represents a change in the interface between Step 2 drivers and the driver loading procedure that does not require changes in driver source code (for example, a change in the DPT produced by a change in the DPTAB macro). The DPTAB macro writes the symbolic constant DPT$K_STEP2_V2 in this field.
DPT$W_DEFUNITS	Number of UCBs that the OpenVMS autoconfiguration facility will automatically create. Drivers specify this number with the **defunits** argument to the DPTAB macro. If the driver also gives a value to DPT$PS_DELIVER, this field is also the number of times that the autoconfiguration facility calls the unit delivery routine. The DPTAB macro writes the value 1 in this field by default.
DPT$W_MAXUNITS	Maximum number of units on controller that this driver supports. Specify this value in the **maxunits** argument to the DPTAB macro. If no value is specified, the default is eight units.
DPT$W_UCBSIZE	Size in bytes of the unit control block for a device that uses this driver. Every driver must specify a value for this field in the **ucbsize** argument to the DPTAB macro. OpenVMS supplies the symbolic constants described in Table 10–17 to represent UCB size. Drivers that employ their own extended UCBs use one of these constants as a base for calculating the size of their extended UCBs. The driver-loading procedure allocates blocks of nonpaged system memory of the specified size when creating UCBs for devices associated with the driver.
DPT$IW_IDB_CRAMS	Number of CRAMS to be allocated and associated with the IDB. The driver-loading procedure allocates the number of CRAMs specified in **idb_crams** argument to the DPTAB macro and inserts them in the linked list headed by IDB$PS_CRAM.
DPT$IW_UCB_CRAMS	Number of CRAMS to be allocated and associated with the IDB. The driver-loading procedure allocates the number of CRAMs specified in **ucb_crams** argument to the DPTAB macro and inserts them in the linked list headed by UCB$PS_CRAM.

(continued on next page)

Table 10–10 (Cont.) Contents of Driver Prologue Table

Field	Use	
DPT$L_FLAGS	Driver-loading flags. The driver can specify any of a set of flags as the value of the **flags** argument to the DPTAB macro. The driver-loading procedure modifies its loading and reloading algorithm based on the settings of these flags. The following bits are defined within DPT$L_FLAGS:	
	DPT$V_SUBCNTRL	Device is a subcontroller.
	DPT$V_SVP	Device requires permanent system page to be allocated during driver loading.
	DPT$V_NOUNLOAD	Driver cannot be reloaded.
	DPT$V_SCS	SCS code must be loaded with this driver.
	DPT$V_DUSHADOW	Driver is the shadowing disk class driver.
	DPT$V_SCSCI	Common SCS/CI subroutines must be loaded with this driver. This bit is ignored on OpenVMS Alpha systems.
	DPT$V_BVPSUBS	Common BVP subroutines must be loaded with this driver. This bit is ignored on OpenVMS Alpha systems.
	DPT$V_UCODE	Driver has an associated microcode image. This bit is ignored on OpenVMS Alpha systems.
	DPT$V_SMPMOD	Driver has been designed to run in an OpenVMS environment.
	DPT$V_DECW_DECODE	Driver is a DECwindows (class input) driver.
	DPT$V_TPALLOC	Select the tape allocation class parameter.
	DPT$V_SNAPSHOT	Driver is certified for system snapshot.
	DPT$V_NO_IDB_DISPATCH	Tells the driver-loading procedure not to create a list of UCB addresses at the end of the IDB (at IDB$L_UCBLST), regardless of the value of the **maxunits** argument to the DPTAB macro or the maximum units specified in the SYSMAN command IO CONNECT.
	DPT$V_SCSI_PORT	Driver is a SCSI port driver.
DPT$IL_ADPTYPE	Type of adapter used by the devices using this driver. The DPTAB macro uses the contents of the **adapter** to construct a symbolic constant of the form AT$_**adapter**, the value of which it inserts in this field.	
DPT$IL_REFC	Number of DDBs that refer to the driver. The driver-loading procedure increments the value in this field each time the procedure creates another DDB that points to the driver's DDT.	

(continued on next page)

Table 10–10 (Cont.) Contents of Driver Prologue Table

Field	Use
DPT$PS_INIT_PD	Procedure value of the driver initialization routine. Every driver must specify a list of values to be written into data structure fields at the time that the driver-loading procedure creates the structures and loads the driver. The driver invokes the DPT_STORE macro once for each value to be written; the macro automatically generates an initialization routine containing code that performs the requested writes, and places its procedure value in this field. The driver-loading procedure calls this initialization routine prior to calling the driver's controller and unit initialization routines.
DPT$PS_REINIT_PD	Procedure value of the driver reinitialization routine. Every driver must specify a list of data structure fields and values to be written into these fields at the time that the driver-loading procedure creates the driver's data structures and loads the driver, or the driver is reloaded. The driver invokes the DPT_STORE macro once for each value to be written; the macro automatically generates a reinitialization routine containing code that performs the requested writes, and places its procedure value in this field. The driver-loading procedure calls the reinitialization routine at driver reloading prior to calling the driver's controller and unit initialization routines. Note that driver reloading is not yet supported on OpenVMS Alpha systems.
DPT$PS_DELIVER_2	Procedure value of the unit delivery routine that the OpenVMS autoconfiguration facility calls once for each of the number of UCBs specified in DPT$W_DEFUNITS. The DPTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **deliver** argument to the macro.
DPT$PS_UNLOAD	Procedure value of the driver routine to be called when driver is reloaded. The DPTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **unload** argument to the macro. The driver-loading procedure calls the driver unloading routine before reinitializing all device units associated with the driver. Note that driver reloading is not yet supported on OpenVMS Alpha systems.
DPT$PS_DDT	Address of DDT.
DPT$PS_DDB_LIST	Header of singly-linked list of DDBs serviced by this driver. This field contains the address of the first DDB in the list. The field DDB$PS_DRVLINK in each DDB points to the next DDB in the list.
DPT$IS_BTORDER	Ordering number for calls to the runtime drivers for boot devices.
DPT$L_VECTOR	Address of a driver-specific vector table. A terminal class or port driver stores the address of its class or port entry vector table in this field. For example, a terminal port driver uses this cell as a pointer to a table of addresses within the driver containing the procedure values of routines in the port driver that are called by the terminal class driver.

(continued on next page)

Table 10–10 (Cont.) Contents of Driver Prologue Table

Field	Use
DPT$T_NAME	Name of the device driver. For each driver, the OpenVMS Alpha driver-loading procedure constructs a 16-byte counted ASCII character string that identifies a driver and stores it in this field. The first byte records the length of the name string; the name string can be up to 15 characters. If you specify the /DRIVER_NAME qualifier in the SYSMAN command IO LOAD or IO CONNECT, the driver-loading procedure generates the name by extracting the filename from the full driver image specification. Otherwise, it creates the driver name from the device name (*ddcu*), appending the string "DRIVER" to the 1 to 9-character device code (*dd*). The driver-loading procedure compares the name of a driver to be loaded with the values in this field in all DPTs already loaded into system memory to ensure that it loads only one copy of a driver at a time.
DPT$L_ECOLEVEL	ECO level of driver, taken from its image header. If for any reason this information is unavailable, the value of this field is left as zero.
DPT$Q_LINKTIME	Time and date at which driver was linked, taken from its image header.
DPT$IQ_IMAGE_NAME	Character string descriptor representing the full file specification of the driver image that has been loaded. To assist the driver loading procedure, this field is initialized as a string descriptor for the entire space available to hold the driver image file specification. The driver loading procedure writes the appropriate descriptor into this field and the driver image file specification in DPT$T_IMAGE_NAME.
DPT$IL_LOADER_ HANDLE	Loader handle for driver image. This field is 16-bytes long and reserved for storing a loadable image handle returned by the loadable executive image loading procedures. When the unloading of loadable executive images is implemented, the handle will be an required input to the unloading mechanism.
DPT$L_UCODE	Address of associated microcode image, if DPT$V_UCODE is set in DPT$L_FLAGS. Use of this field is reserved to Digital.
DPT$L_DECW_SNAME	Offset to a counted ASCII string that allows the SET TERMINAL/SWITCH DCL command to locate an alternate or extension DECwindows class input (decoder) driver.
DPT$Q_LMF_1	First of eight quadwords reserved to Digital for the use of the OpenVMS license management facility. (The others are DPTQ_LMF_2, DPTQ_LMF_3, DPTQ_LMF_4, DPTQ_LMF_5, DPTQ_LMF_6, DPTQ_LMF_7, and DPT$Q_LMF_8.)
DPT$T_IMAGE_NAME	Full file specification of the driver image. This field is NAM$C_MAXRSS long. The driver loading procedure inserts the file specification in DPT$T_IMAGE_NAME, and the character string representing it in DPT$IQ_IMAGE_NAME, when it loads the driver image.

10.9 IDB (Interrupt Dispatch Block)

The interrupt dispatch block (IDB) records controller characteristics. The driver-loading procedure creates and initializes this block when the procedure creates a CRB. The IDB supplies the physical address of the device control and status register (CSR) to the system routines that calculate the values that initialize I/O mailboxes, thus allowing device drivers to access device interface registers.

Table 10–11 describes the interrupt dispatch block.

Table 10–11 Contents of Interrupt Dispatch Block

Field	Use
IDB$Q_CSR	Physical address of the device control and status register (CSR). IOC$CRAM_CMD uses the CSR address in calculations that set up driver transactions to and from I/O space by means of hardware I/O mailboxes. When provided with the address of a device's CSR (for instance, in the SYSMAN command IO CONNECT), the driver-loading procedure writes the specified value into this field. The driver-loading procedure does not test the value before writing this field. For remote DSA devices and local pseudo-devices that require SCS (DPT$IL_ADPTYPE equals AT$_NULL and DPT$V_SCS set in DPT$L_FLAGS), the driver-loading procedure writes a specified SYSID into this field.
IDB$W_SIZE	Size of IDB in bytes. The driver-loading procedure determines the size of the IDB by calculating the size of the ISB$L_UCBLST field and adding it to the symbolic constant IDB$K_BASE_LENGTH. It writes this sum to IDB$W_SIZE when it creates the IDB.
IDB$B_TYPE	Type of data structure. The driver-loading procedure writes the symbolic constant DYN$C_IDB into this field when it creates the IDB.
IDB$W_UNITS	Maximum number of units connected to the controller. The maximum number of units is specified in the **defunits** argument to the DPTAB macro and stored in DPT$W_MAXUNITS. (The default is 8.) This value can be overridden at driver-loading time by the /MAX_UNITS qualifier to the SYSMAN command IO CONNECT. The driver-loading procedure uses this value to determine the size of the IDB$L_UCBLST field.
IDB$B_TT_ENABLE	Reserved for use by terminal port drivers.

(continued on next page)

Table 10–11 (Cont.) Contents of Interrupt Dispatch Block

Field	Use
IDB$PS_OWNER	Address of UCB of device that owns controller data channel. IOC$PRIMITIVE_REQCHANH and IOC$PRIMITIVE_REQCHANL write a UCB address into this field when the routine allocates a controller data channel to a driver. IOC$RELCHAN confirms that the proper driver fork process is releasing a channel by comparing the driver's UCB with the UCB stored in the IDB$PS_OWNER field. If the UCB addresses are the same, IOC$RELCHAN allocates the channel to a waiting driver by writing a new UCB address into the field. If no driver fork processes are waiting for the channel, IOC$RELCHAN clears the field. If the controller is a single-unit controller, the unit or controller initialization routine should write the UCB address of the single device into this field.
IDB$PS_CRAM	Header of singly linked list of CRAMs allocated to the device controller. This field contains the address of the first CRAM in the list. The field CRAM$L_FLINK in each CRAM points to the next CRAM in the list.
IDB$PS_SPL	Address of device lock. The driver-loading procedure copies the value of CRB$PS_DLCK to this field.
IDB$L_ADP	Address of the ADP associated with the device controller. The SYSMAN command IO CONNECT must specify the nexus number of the I/O adapter used by a device. The driver-loading procedure writes the address of the ADP for the specified I/O adapter into the IDB$L_ADP field.
IDB$L_FLAGS	The following bits are defined within IDB$L_FLAGS: IDB$V_CRAM_ALLOC — The driver-loading procedure has allocated the number of CRAMs specified by DPT$IW_IDB_CRAMS and has placed them in the linked list headed by IDB$PS_CRAM. IDB$V_VLE — IDB$L_VECTOR points to a vector list extension (VLE)
IDB$L_DEVICE_SPECIFIC	Longword field available to drivers for device-specific purposes.

(continued on next page)

Table 10–11 (Cont.) Contents of Interrupt Dispatch Block

Field	Use
IDB$L_VECTOR	Offset of interrupt vector for this device controller, or, if IDB$V_VLE in IDB$L_VECTOR is set, the address of a vector list extension (VLE). For device controllers utilizing a single interrupt vector, the driver-loading procedure writes a value into this field using either the autoconfiguration database or the value specified in the /VECTOR qualifier to the SYSMAN command IO CONNECT. This value is a byte offset to device controller's vector location either in the SCB or the ADP vector table. For device controllers utilizing multiple interrupt vectors, the driver-loading procedure writes the address of a vector list extension (VLE) in this field. The field VLE$L_VECTOR_LIST in the VLE contains an array of unsigned longwords, each of which contains a byte offset to a vector location either in the SCB or the ADP vector table. Drivers for devices that utilize programmable interrupt vectors (that is, devices that define their interrupt vector addresses through device registers) must use this field (and, possibly, the contents of VLE$L_VECTOR_LIST) to load those registers during unit initialization and reinitialization after a power failure.
IDB$L_UCBLST	List of UCB addresses. The size of this field is the maximum number of units supported by the controller, as defined in the DPT. The maximum specified in the DPT can be overridden at driver load time by the /MAX_UNITS qualifier to the SYSMAN command IO CONNECT. The driver-loading procedure writes a UCB address at the end of the list located at this symbolic offset in the IDB every time it creates a new UCB associated with the controller.

10.10 IRP (I/O Request Packet)

When a user process queues a valid I/O request by issuing a $QIO or $QIOW system service, the service creates an I/O request packet (IRP). The IRP contains a description of the request and receives the status of the I/O processing as it proceeds.

The I/O request packet is described in Table 10–12. Note that the the standard IRP is followed by fields required by system multiprocessing code and the OpenVMS class drivers. Under no circumstances should a driver not supplied by Digital use these fields.

Table 10–12 Contents of I/O Request Packet (IRP)

Field	Use
IRP$L_IOQFL	I/O queue forward link. EXE$INSERTIRP reads and writes this field when the routine inserts IRPs into a pending-I/O queue. IOC$REQCOM reads and writes this field when the routine dequeues IRPs from a pending-I/O queue in order to send an IRP to a device driver.
IRP$L_IOQBL	I/O queue backward link. EXE$INSERTIRP and IOC$REQCOM read and write these fields.
IRP$W_SIZE	Size of IRP. EXE$QIO writes the symbolic constant IRP$K_LENGTH into this field when the routine allocates and fills an IRP.
IRP$B_TYPE	Type of data structure. EXE$QIO writes the symbolic constant DYN$C_IRP into this field when the routine allocates and fills an IRP.
IRP$B_RMOD	Information used by I/O postprocessing. This field contains the same bit fields as the ACB$B_RMOD field of an AST control block. For instance, the two bits defined at ACB$V_MODE indicate the access mode of the process at time of the I/O request. EXE$QIO obtains the processor access mode from the PS and writes the value into this field.
IRP$L_PID	Process identification of the process that issued the I/O request. EXE$QIO obtains the process identification from the PCB and writes the value into this field.
IRP$L_AST	Procedure value of AST routine, if specified by the process in the I/O request. (This field is otherwise clear.) If the process specifies an AST routine address in the $QIO call, EXE$QIO writes the address in this field. During I/O postprocessing, the special kernel-mode AST routine queues a mode-of-caller AST to the requesting process if this field contains the address of an AST routine.
IRP$L_ASTPRM	Parameter sent as an argument to the AST routine specified by the user in the I/O request. If the process specifies an AST routine and a parameter to that AST routine in the $QIO call, EXE$QIO writes the parameter in this field. During I/O postprocessing, the special kernel-mode AST routine queues a mode-of-caller AST if the IRP$L_AST field contains an address, and passes the value in IRP$L_ASTPRM to the AST routine as an argument.
IRP$L_OBOFF	Original byte offset into the first page of a direct-I/O transfer. For segmented I/O transfers, I/O postprocessing must recalculate the value of IRP$L_BOFF before transferring each segment to account for the difference between the large OpenVMS Alpha memory page size and the 512-byte OpenVMS disk block size. FDT routines store the original byte offset in IRP$L_OBOFF (as well as in IRP$L_BOFF) so that that I/O postprocessing can use IRP$L_OBOFF in conjunction with IRP$L_OBCNT and IRP$L_SVAPTE to unlock the buffer pages locked for the entire transfer.

(continued on next page)

Table 10–12 (Cont.) Contents of I/O Request Packet (IRP)

Field	Use
IRP$L_WIND	Address of window control block (WCB) that describes the file being accessed in the I/O request. EXE$QIO writes this field if the I/O request refers to a file-structured device. An ACP or XQP reads this field. When a process gains access to a file on a file-structured device or creates a logical link between a file and a process I/O channel, the device ACP or XQP creates a WCB that describes the virtual-to-logical mapping of the file data on the disk. EXE$QIO stores the address of this WCB in the IRP$L_WIND field.
IRP$L_UCB	Address of UCB for the device assigned to the I/O channel assigned to the process. EXE$QIO copies this value from the CCB.
IRP$B_EFN	Event flag number and group specified in I/O request. If the I/O request call does not specify an event flag number, EXE$QIO uses event flag 0 by default. EXE$QIO writes this field. The I/O postprocessing routine calls SCH$POSTEF to set this event flag when the I/O operation is complete.
IRP$B_PRI	Base priority of the process that issued the I/O request. EXE$QIO obtains a value for this field from the process control block (PCB). EXE$INSERTIRP reads this field to insert an IRP into a priority-ordered pending-I/O queue.
IRP$B_CLN_INDEX	Shadow clone membership index. Use of this field is reserved to Digital.
IRP$B_SHD_FLAGS	Shadow clone flags. Use of this field is reserved to Digital.
IRP$L_IOSB	Virtual address of the process's I/O status block (IOSB) that receives final status of the I/O request at I/O completion. EXE$QIO writes a value into this field if the I/O request call specifies an IOSB address. (This field is otherwise clear.) The I/O postprocessing special kernel-mode AST routine writes two longwords of I/O status into the IOSB after the I/O operation is complete. When an FDT routine aborts an I/O request by calling EXE$ABORTIO, EXE$ABORTIO fills the IRP$L_IOSB field with zeros so that I/O postprocessing does not write status into the IOSB.
IRP$L_CHAN	Index number of process I/O channel for request. EXE$QIO writes this field.
IRP$L_EXTEND	Address of first IRPE, if any, linked to this IRP. FDT routines write an extension address to this field when a device requires more context than the IRP can accommodate. This field is read by IOC$IOPOST. IRP$V_EXTEND in IRP$L_STS is set if this extension address is used.

(continued on next page)

Table 10–12 (Cont.) Contents of I/O Request Packet (IRP)

Field	Use	
IRP$L_STS	Status of I/O request. EXE$QIO initializes this field to 0. EXE$QIO, FDT routines, driver fork processes, or driver kernel processes modify this field according to the current status of the I/O request. I/O postprocessing reads this field to determine what sort of postprocessing is necessary (for example, deallocate system buffers and adjust quota usage). Bits in the IRP$L_STS field describe the type of I/O function, as follows:	
	IRP$V_BUFIO	Buffered-I/O function
	IRP$V_FUNC	Read function
	IRP$V_PAGIO	Paging-I/O function
	IRP$V_COMPLX	Complex-buffered-I/O function
	IRP$V_VIRTUAL	Virtual-I/O function
	IRP$V_CHAINED	Chained-buffered-I/O function
	IRP$V_SWAPIO	Swapping-I/O function
	IRP$V_DIAGBUF	Diagnostic buffer is present
	IRP$V_PHYSIO	Physical-I/O function
	IRP$V_TERMIO	Terminal I/O (for priority increment calculation)
	IRP$V_MBXIO	Mailbox-I/O function
	IRP$V_EXTEND	An extended IRP is linked to this IRP
	IRP$V_FILACP	File ACP I/O
	IRP$V_MVIRP	Mount-verification I/O function
	IRP$V_SRVIO	Server-type I/O
	IRP$V_KEY	Encrypted function (encryption key address at IRP$L_KEYDESC)
IRP$L_STS2	Second longword of I/O request status. EXE$QIO initializes this field to 0. EXE$QIO, FDT routines, and driver fork processes modify this field according to the current status of the I/O request. Bits in the IRP$L_STS2 field describe the type of I/O function, as follows:	
	IRP$V_START_PAST_HWM	I/O starts past file highwater mark.
	IRP$V_END_PAST_HWM	I/O ends past file highwater mark.
	IRP$V_ERASE	Erase I/O function.
	IRP$V_PART_HWM	Partial file highwater mark update.
	IRP$V_LCKIO	Locked I/O request, as used by DECnet direct I/O.
	IRP$V_SHDIO	Shadowing IRP.
	IRP$V_CACHEIO	I/O using VBN cache buffers.

(continued on next page)

Table 10–12 (Cont.) Contents of I/O Request Packet (IRP)

Field	Use
IRP$L_SVAPTE	For a *direct-I/O* transfer, virtual address of the first page-table entry (PTE) of the I/O-transfer buffer, written here by the FDT routine locking process pages; for a *buffered-I/O* transfer, address of a buffer in system address space, written here by the FDT routine allocating buffer.
	IOC$INITIATE copies this field into UCB$L_SVAPTE before transferring control to a device driver start-I/O routine.
	I/O postprocessing uses this field to deallocate the system buffer for a buffered-I/O transfer or to unlock pages locked for a direct-I/O transfer.
IRP$L_BCNT	Byte count of the I/O transfer. FDT routines calculate the count value and write the field. IOC$INITIATE copies the contents of this field into UCB$L_BCNT before calling a device driver's start-I/O routine.
	For a buffered-I/O-read function, I/O postprocessing uses IRP$L_BCNT to determine how many bytes of data to write to the user's buffer.
IRP$L_BOFF	Byte offset into the first (or current) page of a direct-I/O transfer. FDT routines calculate this offset and write its value into this field and IRP$L_OBOFF. For a segmented direct-I/O transfer, I/O postprocessing recalculates the value of IRP$L_BOFF before transferring each segment to account for difference between the large OpenVMS Alpha memory page size and the 512-byte disk block size.
	For buffered-I/O transfers, FDT routines must write the number of bytes to be charged to the process in this field because these bytes are being used for a system buffer.
	IOC$INITIATE copies this field into UCB$L_BOFF before calling a device driver start-I/O routine.
	I/O postprocessing uses IRP$L_BOFF in conjunction with IRP$L_BCNT and IRP$L_SVAPTE to unlock pages locked for non-segmented direct I/O transfers. For buffered I/O, I/O postprocessing adds the value of IRP$L_BOFF to the process byte count quota.
IRP$PS_KPB	Address of kernel process block (KPB). EXE$KP_ALLOCATE_KPB, when called by EXE$KP_STARTIO, returns the address of the KPB it has allocated to this field.
IRP$L_IOST1	First I/O status longword. IOC$REQCOM and EXE$FINISHIO(C) write the contents of R0 into this field. The I/O postprocessing routine copies the contents of this field into the user's IOSB.
	EXE$ZEROPARM copies a 0 and EXE$ONEPARM copies **p1** into this field. This field, also known as IRP$L_MEDIA, is a good place to put a $QIO request argument. Note that, when error logging is enabled, the contents of IRP$L_MEDIA is copied into an EMB as the "disk size".

(continued on next page)

Table 10–12 (Cont.) Contents of I/O Request Packet (IRP)

Field	Use
IRP$L_IOST2	Second I/O status longword. IOC$REQCOM, EXE$FINISHIO, and EXE$FINISHIO(C) write the contents of R1 into this field. The I/O postprocessing routine copies the contents of this field into the user's IOSB. The low byte of this field is also known as IRP$B_CARCON. IRP$B_CARCON contains carriage control instructions to the driver. EXE$READ and EXE$WRITE copy the contents of **p4** of the user's I/O request into this field.
IRP$L_ABCNT	Accumulated bytes transferred in virtual I/O transfer. IOC$IOPOST reads and writes this field after a partial virtual transfer.
IRP$L_OBCNT	Original transfer byte count in a virtual I/O transfer. IOC$IOPOST reads this field to determine whether a virtual transfer is complete, or whether another I/O request is necessary to transfer the remaining bytes.
IRP$L_SEGVBN	Virtual block number of the current segment of a virtual I/O transfer. IOC$IOPOST writes this field after a partial virtual transfer.
IRP$L_FUNC	I/O function code that identifies the function to be performed for the I/O request. The I/O request call specifies an I/O function code; EXE$QIO and driver FDT routines map the code value to its most basic level (virtual → logical → physical) and copy the reduced value into this field. Based on this function code, EXE$QIO calls FDT action routines to preprocess an I/O request. Six bits of the function code describe the basic function. The remaining 10 bits modify the function. The upper 16 bits of this longword are reserved to Digital.
IRP$L_DIAGBUF	Address of a diagnostic buffer in system address space. If the I/O request call specifies a diagnostic buffer and if a diagnostic buffer length is specified in the DDT, and if the process has diagnostic privilege, EXE$QIO copies the buffer address into this field. EXE$QIO allocates a diagnostic buffer in system address space to be filled by IOC$DIAGBUFILL during I/O processing. During I/O postprocessing, the special kernel-mode AST routine copies diagnostic data from the system buffer into the process diagnostic buffer.
IRP$L_SEQNUM	I/O transaction sequence number. If an error is logged for the request, this field contains the universal error log sequence number.
IRP$L_ARB	Address of access rights block (ARB). This block is located in the PCB and contains the process privilege mask and UIC, which are set up as follows: ARB$Q_PRIV — Quadword containing process privilege mask SPARE$L — Unused longword ARB$L_UIC — Longword containing process UIC
IRP$L_KEYDESC	Address of encryption key.

(continued on next page)

Table 10–12 (Cont.) Contents of I/O Request Packet (IRP)

Field	Use
IRP$L_QIO_P*n*	Function-specific $QIO system service arguments (**p1** through **p6**). EXE$QIO copies these arguments to the appropriate IRP fields.

10.11 IRPE (I/O Request Packet Extension)

I/O request packet extensions (IRPEs) hold additional I/O request information for devices that require more context than the standard IRP can accommodate. IRP extensions are also used when more than one buffer (region) must be locked into memory for a direct-I/O operation, or when a transfer requires a buffer that is larger than 64 KB. An IRPE provides space for two buffer regions, each with a 32-bit byte count.

FDT routines allocate IRPEs by calling EXE$ALLOCIRP. Driver routines link the IRPE to the IRP, store the IRPE's address in IRP$L_EXTEND, and set the bit field IRP$V_EXTEND in IRP$L_STS to show that an IRPE exists for the IRP. The FDT routine initializes the contents of the IRPE. Any fields within the extension not described in Table 10–13 can store driver-dependent information.

If the IRPE specifies additional buffer regions, the FDT routine must explicitly call those buffer locking routines that call back to a driver-specified error routine if the locking procedure fails (EXE$READLOCK_ERR, EXE$WRITELOCK_ERR, and EXE$MODIFYLOCK_ERR). If an error occurs during the locking procedure, the driver must unlock all previously locked regions using MMG$UNLOCK and deallocate the IRPE before returning to the buffer locking routine.

IOC$IOPOST automatically unlocks the pages in region 1 (if defined) and region 2 (if defined) for all the IRPEs linked to the IRP undergoing completion processing. IOC$IOPOST also deallocates all the IRPEs.

The I/O request packet extension is described in Table 10–13.

Table 10–13 Contents of I/O Request Packet Extension (IRPE)

Field	Use
IRPE$W_SIZE	Size of IRPE. EXE$ALLOCIRP writes the constant IRP$K_LENGTH to this field.
IRPE$B_TYPE	Type of data structure. EXE$ALLOCIRP writes the constant DYN$C_IRP to this field.
IRPE$L_EXTEND	Address of next IRPE, if any, for this IRP.
IRPE$L_STS	IRPE status field. If bit IRPE$V_EXTEND is set, it indicates that another IRPE is linked to this one.
IRPE$L_STS2	Second longword of IRPE status field. No bits are currently defined.
IRPE$L_SVAPTE1	System virtual address of the page-table entry (PTE) that maps the start of region 1. FDT routines write this field. If the region is not defined, this field is zero.
IRPE$L_BCNT1	Size in bytes of region 1. FDT routines write this field.

(continued on next page)

Table 10–13 (Cont.) Contents of I/O Request Packet Extension (IRPE)

Field	Use
IRPE$L_BOFF1	Byte offset of region 1. FDT routines write this field.
IRPE$L_SVAPTE2	System virtual address of the PTE that maps the start of region 2. Set by FDT routines. This field contains a value of zero if region 2 is not defined.
IRPE$L_BCNT2	Size in bytes of region 2. FDT routines write this field.
IRPE$L_BOFF2	Byte offset of region 2. This field is set by FDT routines.

10.12 KPB (Kernel Process Block)

The kernel process block (KPB) contains the saved registers, state, and stack pointer for a 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 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.

- Operating system special parameters area

  The operating system 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 is contiguous with either the scheduling or spin lock KPB areas.

- 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 length of each of these areas is rounded to an integral number of quadwords.

The KPB can be used in one of two general types: the OpenVMS executive software type (VEST) and the fully general type (FGT). Typically, OpenVMS software employs 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 which points to the KPB's starting address. By reducing the number of indirect reference operations, accessing VEST KPBs in this manner provides better performance than indirectly accessing the fields in the dynamic portions of a FGT KPB.

You create a VEST KPB by specifying EXE$KP_STARTIO in the **start** argument to the DDTAB macro, or by explicitly invoking KP_ALLOCATE_KPB or calling EXE$KP_ALLOCATE_KPB. Typically VEST KPBs do not include the debugging or parameter areas. If you require either of these areas in a VEST KPB, you must use the KPB allocation macro or routine. When present, the debugging and parameter areas are variable in size and can be located only indirectly through the pointers provided in the base KPB.

In an FGT KPB, only the base KPB and scheduling areas have a fixed position relative to the starting address of the KPB. You can reference fields in either of these areas as offsets from a KPB base pointer register. Because the other KPB areas are variably sized, you can reference them only through the pointers provided in the base KPB.

You create an FGT KPB by explicitly invoking KP_ALLOCATE_KPB or calling EXE$KP_ALLOCATE_KPB. An FGT KPB never includes the OpenVMS special parameters area.

The base, scheduling, OpenVMS special parameters, and spin lock area are described in Table 10–14. Table 10–15 describes the debugging area.

Table 10–14 Contents of Kernel Process Block (KPB)

Field	Use
KPB$PS_FLINK	Forward link. A driver that creates multiple kernel processes can use this field and KPB$PS_BLINK to link together the corresponding KPBs. Doing so facilitates debugging, wherein a determined crash analysis can locate each KPB and associated kernel process stack.
KPB$PS_BLINK	Backward link.
KPB$IW_SIZE	Size of KPB in bytes. For VEST KPBs, EXE$KP_ALLOCATE_KPB writes a value in this field that accounts for the presence of the base KPB, scheduling area, and spin lock area and is rounded up to a quadword multiple.
KPB$IB_TYPE	Type of data structure. EXE$KP_ALLOCATE_KPB writes the symbolic constant DYN$C_MISC in this field when it creates the KPB.
KPB$IB_SUBTYPE	Type of data structure. EXE$KP_ALLOCATE_KPB writes the symbolic constant DYN$C_KPB in this field when it creates the KPB.

(continued on next page)

Table 10–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use	
KPB$IS_STACK_SIZE	Size of kernel process stack in bytes, excluding the two guard pages. EXE$KP_ALLOCATE_KPB computes the size of the kernel process stack by rounding the value of the **stack_size** argument up to an integral number of CPU-specific pages, converting the result to bytes, and storing it in this field. Note that EXE$KP_STARTIO, prior to calling EXE$KP_ALLOCATE_KPB, determines the size of the stack as the maximum of the value of DDT$IS_STACK_BCNT or the symbolic constant KPB$K_MIN_IO_STACK (currently 8KB), rounded up to a multiple of CPU-specific pages.	
KPB$IS_FLAGS	The following bits are defined within KPB$IS_FLAGS.	
	KPB$V_VALID	KPB is valid. EXE$KP_START sets this bit; EXE$KP_END clears it.
	KPB$V_ACTIVE	KPB is in active use. EXE$KP_START sets this bit; EXE$KP_END clears it. EXE$KP_STALL_GENERAL clears this bit when suspending a kernel process; EXE$KP_RESTART sets it when resuming the kernel process.
	KPB$V_VEST	KPB is a VEST KPB. EXE$KP_ALLOCATE_KPB sets this bit in VEST KPBs.
	KPB$V_DELETING	KPB is being deleted. EXE$KP_DEALLOCATE_KPB sets this bit.
	KPB$V_SCHED	Scheduling area is present. EXE$KP_ALLOCATE_KPB sets this bit in VEST KPBs.
	KPB$V_SPLOCK	Spin lock area is present. EXE$KP_ALLOCATE_KPB sets this bit in VEST KPBs.
	KPB$V_DEBUG	Debug area is present.
	KPB$V_PARAM	Parameter area is present.
	KPB$V_DEALLOC_AT_END	KP_END should call KP_DEALLOCATE_KPB. EXE$KP_ALLOCATE_KPB sets this bit in VEST KPBs.
KPB$PS_SAVED_SP	Previous stack pointer. When a kernel process has been started or resumed, this field contains the value of the SP register when the executing thread is preempted (but after the registers indicated by KPB$IS_REG_MASK have been pushed onto the stack). EXE$KP_STALL_GENERAL restores this value to the SP register when the kernel process is suspended.	

(continued on next page)

Table 10–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use
KPB$IS_REG_MASK	Kernel process register save mask. When a kernel process has been suspended, this field contains a mask of the registers that must be restored when the kernel process is resumed. EXE$KP_STARTIO constructs this mask by merging the driver-specified register save mask (DDT$IS_REG_MASK) with the KPB minimal I/O register mask (KPREG$K_MIN_IO_REG_MASK, which includes R2 through R5; the VAX AP, FP, SP, and PC [registers R12 through R15]; and R26, R27, and R29). Registers R0 and R1; R16 through R25; R28; and R30 and R31 (KPREG$K_ERR_REG_MASK) cannot be saved.
KPB$PS_STACK_BASE	System virtual address of the start of the no-access guard page at the base of the kernel process stack. The kernel process stack grows negatively from this address. EXE$KP_ALLOCATE_KPB writes this field when it allocates the stack.
KPB$PS_STACK_SP	Current kernel process SP at the time of suspension. EXE$KP_STALL_GENERAL saves the current value of the SP register to this field when the kernel process is suspended, and restores to the SP register the value in KPB$PS_SAVED_SP. When the kernel process is started, EXE$KP_START initializes this field with the contents of KPB$PS_STACK_BASE. When a kernel process is resumed, EXE$KP_RESTART restores the value in this field to the SP register.
KPB$PS_SCH_PTR	Address of the KPB scheduling area. EXE$KP_ALLOCATE_KPB writes this field when creating the KPB. The scheduling area is contiguous with the base KPB for both VEST KPBs and FGT KPBs, and starts at offset KPB$PS_SCH_STALL_RTN. If you reference fields in the scheduling area as offsets from the address in this field, you must use the prefix KPBSCH$ in place of KPB$ in the symbolic offsets.
KPB$PS_SPL_PTR	Address of the KPB spin lock area. EXE$KP_ALLOCATE_KPB writes this field when creating the KPB. The spin lock area is contiguous with the base KPB and KPB scheduling area for VEST KPBs, and starts at offset KPB$PS_SPL_STALL_RTN. You must use the address in this field to locate the spin lock area for FGT KPBs, using the prefix KPBSPL$ in place of KPB$ in the symbolic offsets to the spin lock area's fields.
KPB$PS_DBG_PTR	Address of the KPB debugging area. EXE$KP_ALLOCATE_KPB writes this field when creating the KPB. See Table 10–15 for a a description of the KPB debugging area. VEST KPBs do not typically include the debugging area.
KPB$PS_PRM_PTR	Address of the KPB parameter area. EXE$KP_ALLOCATE_KPB writes this field when creating the KPB. VEST KPBs do not typically include the parameter area.

(continued on next page)

Table 10–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use
KPB$IS_PRM_LENGTH	Length of the KPB parameter area, as indicated in the **param_length** argument to EXE$KP_ALLOCATE_KPB. EXE$KP_ALLOCATE_KPB rounds this value up to an integral number of quadwords and writes it to this field. VEST KPBs do not typically include the parameter area.
KPB$PS_SCH_STALL_RTN	Procedure value of the routine that has been requested to suspend the kernel process described by this KPB. A kernel process scheduling stall routine preserves kernel process context not represented on the kernel process stack. It also takes steps that allow the stalled kernel process thread to be resumed at some later time (for instance, by inserting a fork block on a fork queue or by making a timer queue entry). A driver can implicitly specify and invoke a scheduling stall routine by calling one of the following system routines: EXEKP_FORK, EXEKP_FORK_WAIT, IOC$KP_REQCHAN, IOC$KP_WFIKPCH, or IOC$KP_WFIRLCH. (The macros KP_STALL_FORK, KP_STALL_FORK_WAIT, KP_STALL_IOFORK, KP_STALL_REQCHAN, KP_STALL_WFIKPCH, and KP_STALL_WFIRLCH may be used to call these routines.) All of these routines call EXE$KP_STALL_GENERAL, which, in turn, issues a standard call to the appropriate scheduling stall routine. A driver can explicitly specify and invoke a scheduling stall routine by calling EXE$KP_STALL_GENERAL (or invoking the KP_STALL_GENERAL macro).
KPB$PS_SCH_RESTRT_RTN	Procedure value of the routine to be invoked by EXE$KP_RESTART when a stalled kernel process is to be resumed. If the kernel process thread was suspended by EXEKP_FORK, EXEKP_FORK_WAIT, IOC$KP_REQCHAN, IOC$KP_WFIKPCH, or IOC$KP_WFIRLCH, this field contains a zero. A driver can explicitly specify and invoke a scheduling restart routine by calling EXE$KP_STALL_GENERAL (or invoking the KP_STALL_GENERAL macro).
KPB$PS_FKBLK	Fork block address. Kernel process scheduling stall routines use this field to locate the fork block in which the kernel process thread's context is to be stored until it is resumed.
KPB$PS_TQFL	Timer-queue forward link for embedded timer queue entry (TQE). Alternatively, as KPB$PS_FQFL, fork-queue forward link for embedded fork block.
KPB$PS_TQBL	Timer-queue backward link. Alternatively, as KPB$PS_FQBL, fork-queue backward link.

(continued on next page)

Table 10–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use
KPB$IW_TQE_SIZE	Size of embedded TQE in bytes. Alternatively, as KPB$IW_FKB_SIZE, size of embedded fork block in bytes. Before using this section of the KPB as a TQE or fork block, you must write the symbolic constant DYN$C_TQE or DYN$C_FRK, as appropriate, in this field.
KPB$IB_FKB_TYPE	Type of data structure. Before using this section of the KPB as a TQE or fork block, you must write the symbolic constant TQE$K_LENGTH or FKB$K_LENGTH, as appropriate, in this field.
KPB$IB_RQTYPE	Type of TQE, as described in *VMS for Alpha Platforms: Internals and Data Structures*. Before using this section of the KPB as an embedded TQE, you must indicate the TQE type in this field. Alternatively, as KPB$IB_FLCK, this field contains the index of the fork lock that synchronizes access to the embedded fork block. Before using this section of the KPB as an embedded fork block, you must write in this field the symbolic constant (as defined by $SPLCODDEF macro in SYS$LIBRARY:LIB.MLB) for the appropriate spin lock index.
KPB$PS_FPC	Procedure value of routine at which execution resumes when the TQE becomes due or when the OpenVMS fork dispatcher dequeues the fork block. (In the latter case, EXEKP_FORK, EXEKP_IOFORK, and EXE$KP_FORK_WAIT write this field when called to suspend driver execution.)
KPB$Q_FR3	Value to be restored to R3 when the TQE becomes due or when the OpenVMS fork dispatcher dequeues the fork block. (In the latter case, EXEKP_FORK, EXEKP_IOFORK, and EXE$KP_FORK_WAIT write this field when called to suspend driver execution.)
KPB$Q_FR4	Value to be restored to R4 when the TQE becomes due or when the OpenVMS fork dispatcher dequeues the fork block. (In the latter case, EXEKP_FORK, EXEKP_IOFORK, and EXE$KP_FORK_WAIT write this field when called to suspend driver execution.)
KPB$IQ_TIME	Quadword system time at which a particular timer event is to occur.
KPB$PS_UCB	UCB address. EXE$KP_STARTIO initializes this field, which exists only in VEST KPBs. Note that this field is also known as KPB$PS_LKB and contains the LKB address when used in lock manager operations.
KPB$PS_IRP	IRP address. EXE$KP_STARTIO initializes this field, which exists only in VEST KPBs.
KPB$IS_TIMEOUT_TIME	Timeout for wait-for-interrupt operation. IOC$KP_WFIKPCH and IOC$KP_WFIRLCH initialize this field, which is used by the corresponding scheduling stall routine when calling the appropriate basic OpenVMS suspension routine. Note that this field exists only in VEST KPBs.

(continued on next page)

Table 10–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use
KPB$IS_RESTORE_IPL	IPL to be restored, and at which execution is to resume, when IOC$KP_WFIKPCH or IOC$KP_WFIRLCH returns to the initiator of the kernel process (that is, the caller of EXE$KP_START or EXE$KP_RESTART). IOC$KP_WFIKPCH and IOC$KP_WFIRLCH initialize this field, which is used by the corresponding scheduling stall routine when calling the appropriate basic OpenVMS suspension routine. Note that this field exists only in VEST KPBs.
KPB$IS_CHANNEL_DATA	Channel data passed to the request-channel scheduling stall routine (by IOC$KP_REQCHAN) and to the wait-for-interrupt scheduling stall routine (by IOC$KP_WFIKPCH or IOC$KP_WFIRLCH) to determine which basic OpenVMS suspension routine to call. Note that only VEST KPBs contain this field. VMS defines the following symbolic constants for this field: KPB$K_KEEP: Keep channel as part of wait-for-interrupt operation (that is, call IOC$PRIMITIVE_WFIKPCH). KPB$K_RELEASE: Release channel as part of wait-for-interrupt operation (that is, call IOC$PRIMITIVE_WFIRLCH). KPB$K_LOW: Insert fork block of UCB requesting controller channel at the tail of the channel-wait queue. KPB$K_HIGH: Insert fork block of UCB requesting controller channel at the head of the channel-wait queue.
KPB$PS_SCSI_PTR1	Generic parameter passing field written and read by SCSI port and class drivers. Note that this field exists only in VEST KPBs.
KPB$PS_SCSI_PTR2	Another generic parameter passing field written and read by SCSI port and class drivers. Note that this field exists only in VEST KPBs.
KPB$PS_SCSI_SCDRP	Address of SCDRP used in SCSI transfers. Note that this field exists only in VEST KPBs.
KPB$IS_TIMEOUT	Timeout time. Note that this field exists only in VEST KPBs.
KPB$IS_NEWIPL	Location in which the SCSI port drivers save the current IPL when invoking the DEVICELOCK macro to synchronize access to a device's database, and from which they restore IPL when invoking the DEVICEUNLOCK macro. Note that this field exists only in VEST KPBs.

(continued on next page)

Table 10–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use
KPB$PS_DLCK	Address of controller's device lock which synchronizes access to device registers and those fields in the UCB accessed at device IPL. SCSI port drivers initialize this field from SPDT$L_DLCK and supply it as the **lockaddr** argument when invoking the DEVICELOCK and DEVICEUNLOCK macros. Note that this field exists only in VEST KPBs.
KPB$PS_SPL_STALL_RTN	Reserved.
KPB$PS_SPL_RESTRT_RTN	Reserved.

Table 10–15 Contents of KPB Debug Area

Field	Use
KPBDBG$IS_START_TIME	Time at which the kernel process was started or last restarted.
KPBDBG$IS_START_COUNT	Number of times the kernel process has been started.
KPBDBG$IS_RESTART_COUNT	Number of times the kernel process has been restarted.
KPBDBG$IS_VEC_INDEX	PC vector index. Indicates which longword in the PC vector index is next to be written
KPBDBG$IS_PC_VEC	Last eight PCs which started, restarted, or suspended the kernel process.

10.13 ORB (Object Rights Block)

The object rights block (ORB) is a data structure that describes the rights a process must have to access the object with which the ORB is associated.

The ORB is usually allocated when the device is connected by means of a SYSMAN IO CONNECT command. The driver loading procedure also sets the address of the ORB in UCB$L_ORB at that time.

The object rights block is described in Table 10–16.

Table 10–16 Contents of Object Rights Block

Field	Use
ORB$L_OWNER	UIC of the object's owner.
ORB$L_ACL_MUTEX	Mutex for the object's access control list (ACL), used to control access to the ACL for reading and writing. The driver-loading procedure initializes this field with –1.
ORB$W_SIZE	Size of ORB in bytes. The driver-loading procedure writes the symbolic constant ORB$K_LENGTH into this field when it creates an ORB.

(continued on next page)

Table 10–16 (Cont.) Contents of Object Rights Block

Field	Use	
ORB$B_TYPE	Type of data structure. The driver-loading procedure writes the symbolic constant DYN$C_ORB into this field when it creates an ORB.	
ORB$B_FLAGS	Flags needed for interpreting portions of the ORB that can have alternate meanings. The following fields are defined within ORB$B_FLAGS:	
	ORB$V_PROT_16	The driver-loading procedure sets this bit to 1, signifying UIC-based protection for this object
	ORB$V_ACL_QUEUE	This flag represents the existence of an ACL queue. The driver-loading procedure does not set this bit.
	ORB$V_MODE_ VECTOR	Use vector mode protection, not byte mode.
	ORB$V_NOACL	This object cannot have an ACL.
	ORB$V_CLASS_PROT	Security classification is valid.
ORB$W_REFCOUNT	Reference count.	
ORB$Q_MODE_PROT	Mode protection vector. The low longword of this quadword is known as ORB$L_MODE.	
ORB$L_SYS_PROT	System protection field. The low word of this field is known as ORB$W_PROT and contains the standard SOGW protection.	
ORB$L_OWN_PROT	Owner protection field.	
ORB$L_GRP_PROT	Group protection field.	
ORB$L_WOR_PROT	World protection field.	
ORB$L_ACLFL	ACL queue forward link. If ORB$V_ACL_QUEUE is 0, this field should contain 0. This field is also known as ORB$L_ACL_COUNT and is cleared by the driver-loading procedure.	
ORB$L_ACLBL	ACL queue backward link. If ORB$V_ACL_QUEUE is 0, this field should contain 0. This field is also known as ORB$L_ACL_DESC and is cleared by the driver-loading procedure.	

10.14 UCB (Unit Control Block)

The unit control block (UCB) is a variable-length block that describes a single device unit. Each device unit on the system has its own UCB. The UCB describes or provides pointers to the device type, controller, driver, device status, and current I/O activity.

During autoconfiguration, the driver-loading procedure creates one UCB for each device unit in the system. A privileged system user can request the driver-loading procedure to create UCBs for additional devices with the SYSMAN command IO CONNECT. The procedure creates UCBs of the length specified in the DPT. The driver uses UCB storage located beyond the standard UCB fields for device-specific data and Step 1 driver storage.

The driver-loading procedure initializes some static UCB fields when it creates the block. OpenVMS and device drivers can read and modify all nonstatic fields of the UCB. The UCB fields that are present for all devices are described in Table 10–18. The length of the basic UCB is defined by the symbol UCB$K_LENGTH.

UCBs are variable in length depending on the type of device and whether the driver performs error logging for the device. OpenVMS defines a number of UCB extensions in the data structure definition macro $UCBDEF and defines a terminal device extension in $TTYUCBDEF. Table 10–17 lists those extensions that are most often used by device drivers, indicating where each is described in this chapter. Note that use of the dual-path extension is reserved to Digital; its contents should remain zero.

Table 10–17 UCB Extensions and Sizes Defined in $UCBDEF

Extension	Used by	Size	Table
Base UCB	All devices	UCB$K_SIZE	10–18
Error log extension	All disk and tape devices	UCB$K_ERL_LENGTH	10–19
Dual-path extension	Reserved to Digital	UCB$K_2P_LENGTH	—
Local tape extension \ 10–20)	All tape devices	UCB$K_LCL_TAPE_LENGTH	value
Local disk extension \ 10–21)	All disk devices	UCB$K_LCL_DISK_LENGTH	value
Terminal extension[1]	Terminal class and port drivers	UCB$K_TT_LENGTH	10–22

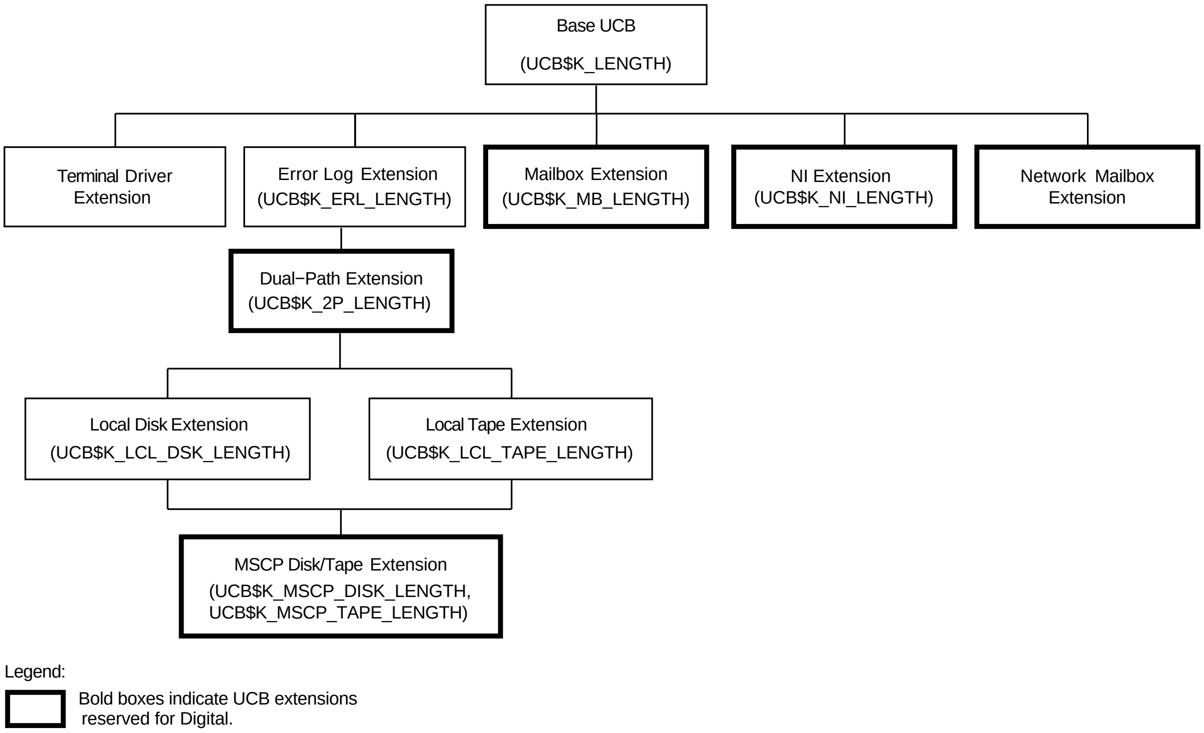
[1] The terminal UCB extension is defined by the data structure definition macro, $TTYUCBDEF.

To use an extended UCB, a device driver must specify its length in the **ucbsize** argument to the DPTAB macro. For instance:

```
DPTAB   -,
    .
    .
    .
        UCBSIZE=UCB$K_LCL_TAPE_LENGTH,-
    .
    .
    .
```

As represented in Figure 10–2, each UCB extension used in a disk or tape driver builds upon the base UCB structure and any extension $UCBDEF defined earlier in the structure. (Note that UCB extensions shown in bold boxes are reserved to Digital.) For instance, if you specify a UCB size of UCB$K_LCL_TAPE_LENGTH, the size of the resulting UCB can accommodate the base UCB, the error log extension, the dual-path extension, and the local tape extension.

Figure 10–2 Composition of Extended Unit Control Blocks



ZK−6620−GE

A device driver can further extend a UCB by using the $DEFINI, $DEF, $DEFEND, and _VIELD macros. For instance:

```
        $DEFINI UCB
.=UCB$K_LCL_DISK_LENGTH

$DEF    UCB$W_XX_FIELD1  .BLKW 1
$DEF    UCB$W_XX_FIELD2  .BLKW 1
$DEF    UCB$L_XX_FLAGS   .BLKL 1
        _VIELD UCB,0,<-
        <XX_BIT1,,M>,-
        <XX_BIT2,,M>,-
        >
$DEF    UCB$K_XX_LENGTH
        $DEFEND UCB
```

In this case, too, the driver must ensure that it specifies the length of the extended UCB in the **ucbsize** argument of the DPTAB macro:

```
DPTAB   -,
        .
        .
        .
        UCBSIZE=UCB$K_XX_LENGTH,-
        .
        .
        .
```

Table 10–18 describes the contents of the unit control block.

Table 10–18 Contents of Unit Control Block

Field	Use
UCB$L_FQFL	Fork queue forward link. The link points to the next entry in the fork queue. EXE$PRIMITIVE_FORK and OpenVMS resource management routines write this field. The queue contains addresses of UCBs that contain driver fork process context of drivers waiting to continue I/O processing.
UCB$L_FQBL	Fork queue backward link. The link points to the previous entry in the fork queue. EXE$PRIMITIVE_FORK and OpenVMS resource management routines write this field.
UCB$W_SIZE	Size of UCB. The DPT of every driver must specify a value for this field. The driver-loading procedure uses the value to allocate space for a UCB and stores the value in each UCB created. Extra space beyond the standard bytes in a UCB (UCB$K_LENGTH) is for device-specific data and Step 1 storage.
UCB$B_TYPE	Type of data structure. The driver-loading procedure writes the constant DYN$C_UCB into this field when the procedure creates the UCB.
UCB$B_FLCK	Index of the fork lock that synchronizes access to this UCB at fork level. The DPT of every driver must specify a value for this field. The driver-loading procedure writes the value in the UCB when the procedure creates the UCB. All devices that are attached to a single I/O adapter and actively compete for shared adapter resources and/or a controller data channel must specify the same value for this field. When the operating system creates a driver fork process to service an I/O request for a device, the fork process gains control at the IPL associated with the fork lock, holding the fork lock itself in a multiprocessing environment. When the driver creates a fork process after an interrupt, OpenVMS inserts the fork block into a processor-specific fork queue based on this fork IPL. A fork dispatcher, executing at fork IPL, obtains the fork lock (if necessary), dequeues the fork block, and restores control to the suspended driver fork process.

(continued on next page)

Table 10–18 (Cont.) Contents of Unit Control Block

Field	Use
UCB$L_FPC	Procedure value of the driver fork routine. When an OpenVMS routine saves driver fork context in order to suspend driver execution, the routine stores the procedure value of the driver entry point at which execution will resume in this field. A system routine that reactivates a suspended driver transfers control to the saved PC address. System routines that suspend driver processing include EXE$PRIMITIVE_FORK, IOC$PRIMITIVE_REQCHANL, IOC$PRIMITIVE_REQCHANH, IOC$PRIMITIVE_WFIKPCH, IOC$PRIMITIVE_WFIRLCH, EXE$KP_STALL_GENERAL, EXEKP_FORK, EXEKP_FORK_WAIT, IOC$KP_REQCHAN, IOC$KP_WFIKPCH, and IOC$KP_WFIRLCH. Routines that reactivate suspended driver routines include IOC$RELCHAN, the OpenVMS fork dispatcher, and driver interrupt service routines. When a driver interrupt service routine determines that a device is expecting an interrupt, the routine restores control to the saved PC address in the device's UCB.
UCB$Q_FR3	Value of R3 at the time that a system routine suspends a driver fork process. The value of R3 is restored just before a suspended driver regains control.
UCB$Q_FR4	Value of R4 at the time that a system routine suspends a driver fork process. The value of R4 is restored just before a suspended driver regains control.
UCB$W_BUFQUO	Buffered-I/O quota if the UCB represents a mailbox.
UCB$W_INIQUO	Initial buffered-I/O quota if the UCB represents a mailbox.
UCB$L_ORB	Address of ORB associated with the UCB. The driver-loading procedure places the address in this field.
UCB$L_LOCKID	Lock management lock ID of device allocation lock. A lock management lock is used for device allocation so that device allocation functions properly for cluster-accessible devices in a VAXcluster (DEV$V_CLU set within UCB$L_DEVCHAR2).
UCB$PS_CRAM	Header of singly linked list of CRAMs allocated to the device unit. This field contains the address of the first CRAM in the list. The field CRAM$L_FLINK in each CRAM points to the next CRAM in the list.
UCB$L_CRB	Address of primary CRB associated with the device. The driver-loading procedure writes this field. Driver fork processes read this field to gain access to device registers. system routines use UCB$L_CRB to locate interrupt-dispatching code and the addresses of driver unit and controller initialization routines.
UCB$L_DLCK	Address of device lock that—in a multiprocessing environment—synchronizes access to device registers and those fields in the UCB accessed at device IPL. The driver-loading routine copies the address of the device lock in the CRB (CRB$PS_DLCK) to this field as it creates a UCB for each device on a controller.

(continued on next page)

Table 10–18 (Cont.) Contents of Unit Control Block

Field	Use
UCB$L_DDB	Address of DDB associated with device. The driver-loading procedure writes this field when the procedure creates the associated UCB. system routines generally read the DDB field in order to locate device driver entry points, the address of a driver FDT, or the ACP associated with a given device.
UCB$L_PID	Process identification number of the process that has allocated the device. Written by the $ALLOC system service.
UCB$L_LINK	Address of next UCB in the chain of UCBs attached to a single controller and associated with a DDB. The driver-loading procedure writes this field when the procedure adds the next UCB. Any system routine that examines the status of all devices on the system reads this field. Such routines include EXE$TIMEOUT, IOC$SEARCHDEV, and power failure recovery routines.
UCB$L_VCB	Address of volume control block (VCB) that describes the volume mounted on the device. This field is written by the device's ACP and read by EXE$QIOACPPKT, ACPs, and the XQP.
UCB$L_DEVCHAR	First longword of device characteristics bits. The DPT of every driver should specify symbolic constant values (defined by the $DEVDEF macro in SYS$LIBRARY:STARLET.MLB) for this field. The driver-loading procedure writes the field when the procedure creates the UCB. The $QIO system service reads the field to determine whether a device is spooled, file structured, shared, has a volume mounted, and so on. The system defines the following device characteristics:

DEV$V_REC	Record-oriented device
DEV$V_CCL	Carriage control device
DEV$V_TRM	Terminal device
DEV$V_DIR	Directory-structured device
DEV$V_SDI	Single directory-structured device
DEV$V_SQD	Sequential block-oriented device (magnetic tape, for example)
DEV$V_SPL	Device spooled
DEV$V_OPR	Operator device
DEV$V_RCT	Device contains RCT
DEV$V_NET	Network device
DEV$V_FOD	File-oriented device (disk and magnetic tape, for example)
DEV$V_DUA	Dual-ported device
DEV$V_SHR	Shareable device (used by more than one program simultaneously)
DEV$V_GEN	Generic device

(continued on next page)

Table 10–18 (Cont.) Contents of Unit Control Block

Field	Use	
	DEV$V_AVL	Device available for use
	DEV$V_MNT	Device mounted
	DEV$V_MBX	Mailbox device
	DEV$V_DMT	Device marked for dismount
	DEV$V_ELG	Error logging enabled
	DEV$V_ALL	Device allocated
	DEV$V_FOR	Device mounted as foreign (not file structured)
	DEV$V_SWL	Device software write-locked
	DEV$V_IDV	Device capable of providing input
	DEV$V_ODV	Device capable of providing output
	DEV$V_RND	Device allowing random access
	DEV$V_RTM	Real-time device
	DEV$V_RCK	Read-checking enabled
	DEV$V_WCK	Write-checking enabled
UCB$L_DEVCHAR2	Second longword of device characteristics. The DPT of every driver should specify symbolic constant values (defined by the $DEVDEF macro in SYS$LIBRARY:STARLET.MLB) for this field. The driver-loading procedure writes the field when the procedure creates the UCB. The system defines the following device characteristics:	
	DEV$V_CLU	Device available clusterwide
	DEV$V_DET	Detached terminal
	DEV$V_RTT	Remote-terminal UCB extension
	DEV$V_CDP	Dual-pathed device with two UCBs
	DEV$V_2P	Two paths known to device
	DEV$V_MSCP	Disk or tape accessed using MSCP
	DEV$V_SSM	Shadow set member
	DEV$V_SRV	Served by MSCP server
	DEV$V_RED	Redirected terminal
	DEV$V_NNM	Device name has a prefix of the format "*node*$"
	DEV$V_WBC	Device supports write-back caching
	DEV$V_WTC	Device supports write-through caching
	DEV$V_HOC	Device supports host caching
	DEV$V_LOC	Device accessible via local (non-emulated) controller
	DEV$V_DFS	Device is DFS-served
	DEV$V_DAP	Device is DAP accessed

(continued on next page)

Table 10–18 (Cont.) Contents of Unit Control Block

Field	Use	
	DEV$V_NLT	Device has no bad block information on its last track
	DEV$V_SEX	Device (TAPE) supports serious exception handling
	DEV$V_SHD	Device is a member of a host based shadow set
	DEV$V_VRT	Device is a shadow set virtual unit
	DEV$V_LDR	Loader present (tapes)
	DEV$V_NOLB	Device ignores server load balancing requests
	DEV$V_NOCLU	Device will never be available clusterwide
	DEV$V_VMEM	Virtual member of a constituent set
	DEV$V_SCSI	Device is a SCSI device
	DEV$V_WLG	Device has write logging capability
	DEV$V_NOFE	Device does not support forced error
UCB$L_AFFINITY	Bit mask of the CPU IDs of processors in an OpenVMS multiprocessing system that have physical connectivity to the device. Such processors can thereby access the device's registers and initiate I/O operations on the device.	
UCB$L_XTRA	Extra longword for SMP. This field is also known as UCB$L_ALTIOWQ (alternate start-I/O request wait queue).	
UCB$B_DEVCLASS	Device class. The DPT of every driver should specify a symbolic constant (defined by the $DCDEF macro in SYS$LIBRARY:STARLET.MLB) for this field. The driver-loading procedure writes this field when it creates the UCB. Drivers with set mode and device characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request. VMS defines the following device classes:	
	DC$_DISK	Disk
	DC$_TAPE	Tape
	DC$_SCOM	Synchronous communications
	DC$_CARD	Card reader
	DC$_TERM	Terminal
	DC$_LP	Line printer

(continued on next page)

Table 10–18 (Cont.) Contents of Unit Control Block

Field	Use	
	DC$_WORKSTATION	Workstation
	DC$_REALTIME	Real time. Note that the definition of a device as a real-time device (DC$_REALTIME) is somewhat subjective; it implies no special treatment by OpenVMS.
	DC$_BUS	Bus
	DC$_MAILBOX	Mailbox
	DC$_REMCSL_STORAGE	Remote console storage
	DC$_MISC	Miscellaneous
UCB$B_DEVTYPE	Device type. The DPT of every driver should specify a symbolic constant (defined by the $DCDEF macro in SYS$LIBRARY:STARLET.MLB) for this field. The driver-loading procedure writes the field when it creates the UCB.	
	Drivers for devices with set mode and set characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request.	
UCB$W_DEVBUFSIZ	Default buffer size. The DPT can specify a value for this field if relevant. The driver-loading procedure writes the field when it creates the UCB.	
	Drivers for devices with set mode and set characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request. This field is used by RMS for record I/O on nonfile devices.	
UCB$Q_DEVDEPEND	Device-descriptive data interpreted by the device driver itself. The DPT can specify a value for this field. The driver-loading procedure writes this field when it creates the UCB.	
	Drivers for devices with set mode and set characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request.	
UCB$Q_DEVDEPND2	Second quadword for device-dependent status. This field is an extension of UCB$Q_DEVDEPEND.	
UCB$L_IOQFL	Pending-I/O queue listhead forward link. The queue contains the addresses of IRPs waiting for processing on a device. EXE$INSERTIRP inserts IRPs into the pending-I/O queue when a device is busy. IOC$REQCOM dequeues IRPs when the device is idle.	
	The queue is a priority queue that has the highest priority IRPs at the front of the queue. Priority is determined by the base priority of the requesting process. IRPs with the same priority are processed first-in/first-out.	

(continued on next page)

Table 10–18 (Cont.) Contents of Unit Control Block

Field	Use
UCB$L_IOQBL	Pending-I/O queue listhead backward link. EXE$INSERTIRP and IOC$REQCOM modify the pending-I/O queue.
UCB$W_UNIT	Number of the physical device unit; stored as a binary value. The driver-loading procedure writes a value into this field when it creates the UCB. Drivers for multiunit controllers read this field during unit initialization to identify a unit to the controller.
UCB$W_CHARGE	Mailbox byte count quota charge, if the device is a mailbox.
UCB$L_IRP	Address of IRP currently being processed on the device unit by the driver fork process. IOC$INITIATE writes the address of an IRP into this field before the routine creates a driver fork process to handle an I/O request. From this field, a driver fork process obtains the address of the IRP being processed. The value contained in this field is not valid if the UCB$V_BSY bit in UCB$L_STS is clear.
UCB$L_REFC	Reference count of processes that currently have process I/O channels assigned to the device. The $ASSIGN and $ALLOC system services increment this field. The $DASSGN and $DALLOC system services decrement this field.
UCB$B_DIPL	Interrupt priority level (IPL) at which the device requests hardware interrupts. The DPT of every driver must specify a value for this field. The driver-loading procedure writes this field when the procedure creates the UCB. When the driver-loading procedure subsequently creates the device lock's spin lock structure (SPL), it moves the contents of this field into SPL$B_IPL. In an OpenVMS multiprocessing environment, drivers obtain the device lock at UCB$L_DLCK before reading or writing device registers or accessing other fields in the UCB synchronized at device IPL, thereby also raising IPL to device IPL in the process.
UCB$B_AMOD	Access mode at which allocation occurred, if the device is allocated. Written by the $ALLOC and $DALLOC system services.
UCB$L_AMB	Associated mailbox UCB pointer. A spooled device uses this field for the address of its associated device. Devices that are nonshareable and not file oriented can use this field for the address of an associated mailbox.

(continued on next page)

Table 10–18 (Cont.) Contents of Unit Control Block

Field	Use	
UCB$L_STS	Device unit status (formerly UCB$W_STS). Written by drivers, IOC$REQCOM, IOC$CANCELIO, IOC$INITIATE, IOC$WFIKPCH, IOC$WFIRLCH, EXE$INSIOQ, and EXE$TIMEOUT. This field is read by drivers, the $QIO system service routines, IOC$REQCOM, IOC$INITIATE, and EXE$TIMEOUT. This longword includes the following bits:	
	UCB$V_TIM	Timeout enabled.
	UCB$V_INT	Interrupts expected.
	UCB$V_ERLOGIP	Error log in progress.
	UCB$V_CANCEL	Cancel I/O on unit.
	UCB$V_ONLINE	Device is on line.
	UCB$V_POWER	Power has failed while unit was busy.
	UCB$V_TIMOUT	Unit is timed out.
	UCB$V_INTTYPE	Receiver interrupt.
	UCB$V_BSY	Unit is busy.
	UCB$V_MOUNTING	Device is being mounted.
	UCB$V_DEADMO	Deallocate device at dismount.
	UCB$V_VALID	Volume appears valid to software.
	UCB$V_UNLOAD	Unload volume at dismount.
	UCB$V_TEMPLATE	Template UCB from which other UCBs for this device are made. The $ASSIGN system service checks this bit in the requested UCB and, if the bit is set, creates a UCB from the template. The new UCB is assigned instead.
	UCB$V_MNTVERIP	Mount verification in progress.
	UCB$V_WRONGVOL	Volume name does not match name in the VCB.
	UCB$V_DELETEUCB	Delete this UCB when the value in UCB$W_REFC becomes zero.
	UCB$V_LCL_VALID	The volume on this device is valid on the local node.
	UCB$V_SUPMVMSG	Suppress mount-verification messages if they indicate success.
	UCB$V_MNTVERPND	Mount verification is pending on the device and the device is busy.

(continued on next page)

Table 10–18 (Cont.) Contents of Unit Control Block

Field	Use	
	UCB$V_DISMOUNT	Dismount in progress.
	UCB$V_CLUTRAN	VAXcluster state transition in progress.
	UCB$V_WRTLOCKMV	Write-locked mount verification in progress.
	UCB$V_SVPN_END	Last byte used from page is mapped by a system virtual page number.
	UCB$V_ALTBSY	Unit is busy via alternate STARTIO path.
	UCB$V_SNAPSHOT	Restart validation is in progress.
UCB$L_DEVSTS	Device-dependent status.	
	The system defines the following status bits:	
	UCB$V_PRMMBX	Device is a permanent mailbox. OpenVMS also defines this bitfield as UCB$V_JOB (job controller has been notified).
	UCB$V_DELMBX	Mailbox is marked for deletion.
	UCB$V_SHMMBX	Device is shared-memory mailbox.
	UCB$V_TEMPL_BSY	Template UCB is busy.
	Disk drivers use bits in UCB$L_DEVSTS as follows:	
	UCB$V_ECC	ECC correction made.
	UCB$V_DIAGBUF	Diagnostic buffer is specified.
	UCB$V_NOCNVRT	No logical block number to media address conversion.
	UCB$V_DX_WRITE	Console floppy write operation.
	UCB$V_DATACACHE	Data blocks are being cached.
	Terminal class and port drivers use bits in UCB$L_DEVSTS as follows:	
	UCB$V_TT_TIMO	Terminal read timeout in progress.
	UCB$V_TT_NOTIF	Terminal user notified of unsolicited data.
	UCB$V_TT_HANGUP	Process hang up.
	UCB$V_TT_NOLOGINS	No logins allowed.
UCB$L_QLEN	Number of entries in pending-I/O queue (pointed to by UCB$L_IOQFL).	

(continued on next page)

Table 10–18 (Cont.) Contents of Unit Control Block

Field	Use
UCB$L_DUETIM	Due time for I/O completion. Stored as the low-order 32-bit absolute time (time in seconds since the operating system was booted) at which the device will time out. IOC$PRIMITIVE_WFIKPCH and IOC$PRIMITIVE_WFIRLCH write this value when they suspend a driver to wait for an interrupt or timeout. EXE$TIMEOUT examines this field in each UCB in the I/O database once per second. If the timeout has occurred and timeouts are enabled for the device, EXE$TIMEOUT calls the device driver timeout handler.
UCB$L_OPCNT	Count of operations completed on device unit since last system bootstrap. IOC$REQCOM writes this field every time the routine inserts an IRP into the I/O postprocessing queue.
UCB$L_SVPN	Index to the virtual address of the system PTE that the driver loading procedure has permanently allocated to the device. The system virtual address of the page described by this index can be calculated by the following formula: (index * PTE$C_BYTES_PER_PTE) + MMG$GL_SPTBASE If a DPT specifies DPT$M_SVP in the **flags** argument to the DPTAB macro, the driver-loading procedure allocates a page of nonpaged system memory to the device. The procedure writes the system PTE's index into UCB$L_SVPN when the procedure creates the UCB. Disk drivers use this field for ECC error correction.
UCB$L_SVAPTE	For a *direct-I/O* transfer, the virtual address of the system PTE for the first page to be used in the transfer; for a *buffered-I/O* transfer, the virtual address of the system buffer used in the transfer. IOC$INITIATE writes this field from IRP$L_SVAPTE before calling a driver start-I/O routine. Drivers read this value to compute the starting address of a transfer.
UCB$L_BCNT	Count of bytes in the I/O transfer. IOC$INITIATE copies this field from the IRP. Drivers read this field to determine how many bytes to transfer in an I/O operation.
UCB$L_BOFF	For a *direct-I/O* transfer, the byte offset into the first page of the transfer buffer; for a *buffered-I/O* transfer, the number of bytes charged to the process for the transfer. IOC$INITIATE copies this field from the IRP. Drivers read the field in calculating the starting address of a DMA transfer. If only part of a DMA transfer succeeds, the driver adjusts the value in this field to be the byte offset in the first page of the data that was not transferred.
UCB$L_SOFTERRCNT	Reserved to Digital.

(continued on next page)

Table 10–18 (Cont.) Contents of Unit Control Block

Field	Use
UCB$L_ERTCNT	Error retry count of the current I/O transfer. The driver sets this field to the maximum retry count each time it begins I/O processing. Before each retry, the driver decreases the value in this field. During error logging, IOC$REQCOM copies the value into the error message buffer.
UCB$L_ERTMAX	Maximum error retry count allowed for single I/O transfer. The DPT of some drivers specifies a value for this field. The driver-loading procedure writes the field when the procedure creates the UCB. During error logging, IOC$REQCOM copies the value into the error message buffer.
UCB$L_ERRCNT	Number of errors that have occurred on the device since system booted. The driver-loading procedure initializes the field to 0 when the procedure creates the UCB. ERL$DEVICERR and ERL$DEVICTMO increment the value in the field and copy the value into an error message buffer. The DCL command SHOW DEVICE displays in its error count column the value contained in this field.
UCB$L_PDT	Address of port descriptor table (PDT) or SCSI port descriptor table (SPD). This field is reserved for OpenVMS SCS and SCSI port drivers.
UCB$L_DDT	Address of DDT for unit. The driver load procedure writes the contents of DDB$L_DDT for the device controller to this field when it creates the UCB.
UCB$PS_ADP	Address of ADP. The driver-loading procedure initializes this field.
UCB$PS_CRCTX	Address of CRCTX. A driver initializes this field when it allocates a CRCTX.
UCB$L_MEDIA_ID	Bit-encoded media name and type, used by MSCP devices.
UCB$PS_DTN	Address of device-type name structure (DTN). Reserved to Digital.

Table 10–19 describes the contents of the UCB error log extension.

Table 10–19 Contents of UCB Error Log Extension

UCB$L_EMB	Address of error message buffer. If error logging is enabled and a device/controller error or timeout occurs, the driver calls ERL$DEVICERR or ERL$DEVICTMO to allocate an error message buffer and copy the buffer address into this field. IOC$REQCOM writes final device status, error counters, and I/O request status into the buffer specified by this field.
UCB$L_FUNC	I/O function modifiers. This field is read and written by drivers that log errors.
UCB$L_DPC	Device-specific field. This field is reserved for driver use.

Table 10–20 describes the contents of th UCB local tape extension.

Table 10–20 Contents of UCB Local Tape Extension

Field Name	Contents
UCB$W_DIRSEQ	Directory sequence number. If the high-order bit of this word, UCB$V_AST_ARMED, is set, it indicates that the requesting process is blocking ASTs.
UCB$B_ONLCNT	Number of times the device has been placed on line since system booted.
UCB$B_PREV_RECORD	Tape position prior to the start of the last I/O operation.
UCB$L_RECORD	Current tape position or frame counter.
UCB$L_TMV_RECORD	Position following last guaranteed successful I/O operation.
UCB$W_TMV_CRC1	First CRC for mount verification's media validation.
UCB$W_TMV_CRC2	Second CRC for mount verification's media validation.
UCB$W_TMV_CRC3	Third CRC for mount verification's media validation.
UCB$W_TMV_CRC4	Fourth CRC for mount verification's media validation.

Table 10–21 describes the contents of the UCB local disk extension.

Table 10–21 Contents of UCB Local Disk Extension

Field Name	Contents
UCB$W_DIRSEQ	Directory sequence number. If the high-order bit of this word, UCB$V_AST_ARMED, is set, it indicates that the requesting process is blocking ASTs.
UCB$B_ONLCNT	Number of times device has been placed on line since OpenVMS was last bootstrapped.
UCB$L_MAXBLOCK	Maximum number of logical blocks on random-access device. This field is written by a disk driver during unit initialization and power recovery.
UCB$L_MAXBCNT	Maximum number of bytes that can be transferred. A disk driver writes this field during unit initialization and power recovery.
UCB$L_DCCB	Pointer to cache control block.
UCB$L_QLENACC	Queue length accumulator.

Table 10–22 describes the contents of the UCB terminal extension.

Table 10–22 Contents of UCB Terminal Extension

Field	Use	
UCB$L_TL_CTRLY	Listhead of CTRL/Y AST control blocks (ACBs).	
UCB$L_TL_CTRLC	Listhead of CTRL/C ACBs.	
UCB$L_TL_OUTBAND	Out-of-band character mask.	
UCB$L_TL_BANDQUE	Listhead of out-of-band ACBs.	
UCB$L_TL_PHYUCB	Address of physical UCB.	
UCB$L_TL_CTLPID	Process ID of controlling process (used with SPAWN).	
UCB$Q_TL_BRKTHRU	Facility broadcast bit mask.	
UCB$L_TL_POSIX_DATA	POSIX PTC pointer	
UCB$L_TL_ASIAN_DATA	Pointer to Asian language data.	
UCB$L_TL_A_CHARSET	Character set bitmask. The lowest byte of this field is also known as UCB$B_TL_A_MODE and represents the current Asian modes.	
UCB$L_TL_A_FI_UCB	Pointer to Asian input server.	
UCB$L_TT_RDUE	Absolute time at which a read timeout is due.	
UCB$L_TT_RTIMOU	Address of read timeout routine.	
UCB$L_TT_STATE1	First longword of terminal state information.	
	The following fields are defined within UCB$L_TT_STATE1:	
	TTY$V_ST_POWER	Power failure
	TTY$V_ST_CTRLS	Class output
	TTY$V_ST_MODEM_OFF	Modem off
	TTY$V_ST_FILL	Fill mode
	TTY$V_ST_CURSOR	Cursor
	TTY$V_ST_SENDLF	Forced line feed
	TTY$V_ST_BACKSPACE	Backspace
	TTY$V_ST_MULTI	Multi-echo
	TTY$V_ST_WRITE	Write in progress
	TTY$V_ST_EOL	End of line
	TTY$V_ST_EDITREAD	Editing read in progress
	TTY$V_ST_RDVERIFY	Read verify in progress
	TTY$V_ST_RECALL	Command recall
	TTY$V_ST_READ	Read in progress
	TTY$V_ST_POSIXREAD	POSIX read
UCB$L_TT_STATE2	Second longword of terminal state information.	
	The following fields are defined within UCB$L_TT_STATE2:	
	TTY$V_ST_CTRLO	Output enable
	TTY$V_ST_DEL	Delete

(continued on next page)

Table 10–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
	TTY$V_ST_PASALL	Pass-all mode
	TTY$V_ST_NOECHO	No echo
	TTY$V_ST_WRTALL	Write-all mode
	TTY$V_ST_PROMPT	Prompt
	TTY$V_ST_NOFLTR	No control-character filtering
	TTY$V_ST_ESC	Escape sequence
	TTY$V_ST_BADESC	Bad escape sequence
	TTY$V_ST_NL	New line
	TTY$V_ST_REFRSH	Refresh
	TTY$V_ST_ESCAPE	Escape mode
	TTY$V_ST_TYPFUL	Type-ahead buffer full
	TTY$V_ST_SKIPLF	Skip line feed
	TTY$V_ST_ESC_O	Output escape
	TTY$V_ST_WRAP	Wrap enable
	TTY$V_ST_OVRFLO	Overflow condition
	TTY$V_ST_AUTOP	Autobaud pending
	TTY$V_ST_CTRLR	Clock prompt and data string from read buffer
	TTY$V_ST_SKIPCRLF	Skip line feed following a carriage return
	TTY$V_ST_EDITING	Editing operation
	TTY$V_ST_TABEXPAND	Expand tab characters
	TTY$V_ST_QUOTING	Quote character
	TTY$V_ST_OVERSTRIKE	Overstrike mode
	TTY$V_ST_TERMNORM	Standard terminator mask
	TTY$V_ST_ECHAES	Alternate echo string
	TTY$V_ST_PRE	Pre-type-ahead mode
	TTY$V_ST_NINTMULTI	Noninterrupt multi-echo mode
	TTY$V_ST_RECONNECT	Reconnect operation
	TTY$V_ST_CTSLOW	Clear-to-send low
	TTY$V_ST_TABRIGHT	Check for tabs to the right of the current position
UCB$L_TT_LOGUCB	Address of logical UCB, if the redirect bit is set (DEV$V_RED in UCB$L_DEVCHAR2). If this UCB describes the logical UCB, the contents of UCB$L_TT_LOGUCB are zero.	
UCB$L_TT_DECHAR	First longword of default device characteristics.	

(continued on next page)

Table 10–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
UCB$L_TT_DECHA1	Second longword of default device characteristics.	
UCB$L_TT_DECHA2	Third longword of default device characteristics.	
UCB$L_TT_DECHA3	Fourth longword of default device characteristics.	
UCB$L_TT_WFLINK	Write queue forward link.	
UCB$L_TT_WBLINK	Write queue backward link.	
UCB$L_TT_WRTBUF	Current write buffer block.	
UCB$L_TT_MULTI	Address of current multi-echo buffer.	
UCB$W_TT_MULTILEN	Length of multi-echo string to be written.	
UCB$W_TT_SMLTLEN	Saved length of multi-echo string.	
UCB$L_TT_SMLT	Saved address of multi-echo buffer.	
UCB$W_TT_DESPEE	Default speed.	
UCB$B_TT_DECRF	Default carriage-return fill.	
UCB$B_TT_DELFF	Default line-feed fill.	
UCB$B_TT_DEPARI	Default parity/character size.	
UCB$B_TT_DETYPE	Default terminal type.	
UCB$W_TT_DESIZE	Default line size.	
UCB$W_TT_SPEED	Terminal line speed. This field is read and written by the class driver, and read by the port driver. It contains the following byte fields:	
	UCB$B_TT_TSPEED	Transmit speed
	UCB$B_TT_RSPEED	Receive speed
UCB$B_TT_CRFILL	Number of fill characters to be output for carriage return.	
UCB$B_TT_LFFILL	Number of fill characters to be output for line feed.	
UCB$B_TT_PARITY	Parity, frame and stop bit information to be set when the PORT_SET_LINE service routine is called. This field is read and written by the class driver, and read by the port driver. It contains the following bit fields:	
	UCB$V_TT_XXPARITY	Reserved to Digital.
	UCB$V_TT_DISPARERR	Reserved to Digital.
	UCB$V_TT_USERFRAME	Reserved to Digital.
	UCB$V_TT_LEN	Two bits signifying character length (not counting start, stop, and parity bits), as follows: 00_2 = 5 bits; 01_2 = 6 bits; 10_2 = 7 bits; and 11_2 = 8 bits.
	UCB$V_TT_STOP	Number of stop bits: clear if one stop bit; set if two stop bits.

(continued on next page)

Table 10–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
	UCB$V_TT_PARTY	Parity checking. This bit is set if parity checking is enabled.
	UCB$V_TT_ODD	Parity type: clear if even parity; set if odd parity.
UCB$L_TT_TYPAHD	Address of type-ahead buffer.	
UCB$W_TT_CURSOR	Current cursor position.	
UCB$B_TT_LINE	Current line position on page.	
UCB$B_TT_LASTC	Last formatted output character.	
UCB$W_TT_BSPLEN	Number of back spaces to output for non-ANSI terminals.	
UCB$B_TT_FILL	Current fill character count.	
UCB$B_TT_ESC	Current read escape syntax state.	
UCB$B_TT_ESC_O	Current write escape syntax state.	
UCB$B_TT_INTCNT	Number of characters in interrupt string.	
UCB$W_TT_UNITBIT	Enable and disable modem control.	
UCB$W_TT_HOLD	Port driver's internal flags and unit holding tank. This is read and written by the port driver, and is not accessed by the class driver. It contains the following subfields:	
	TTY$B_TANK_CHAR	Character.
	TTY$V_TANK_PREMPT	Send preempt character.
	TTY$V_TANK_STOP	Stop output.
	TTY$V_TANK_HOLD	Character stored in TTY$B_TANK_CHAR.
	TTY$V_TANK_BURST	Burst is active.
	TTY$V_TANK_DMA	DMA transfer is active.
UCB$B_TT_PREMPT	Preempt character.	
UCB$B_TT_OUTYPE	Amount of data to be written on a callback from the class driver. When negative, this field indicates that there is a burst of data ready to be returned; when zero, it signifies that no data is to be written; and when 1, it indicates that a single character is to be written. This field is written by the class driver and read by the port driver.	
UCB$L_TT_GETNXT	Address of the class driver's input routine. This field is read by the port driver.	
UCB$L_TT_PUTNXT	Address of the class driver's output routine. This field is read by the port driver.	
UCB$L_TT_CLASS	Address of the class driver's vector table. This field is initialized by the CLASS_CTRL_INIT macro. The port driver reads UCB$L_TT_CLASS whenever it must call the class driver at an entry point other than UCB$L_TT_GETNXT or UCB$L_TT_PUTNXT.	
UCB$L_TT_PORT	Address of the port driver's vector table.	

(continued on next page)

Table 10–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
UCB$L_TT_OUTADR	Address of the first character of a burst of data to be written. This field is only valid when UCB$B_TT_OUTYPE contains –1. It is read and written by the port driver, and written by the class driver.	
UCB$W_TT_OUTLEN	Number of characters in a burst of data to be written. This field is only valid when UCB$B_TT_OUTYPE contains –1. It is read and written by the port driver, and written by the class driver.	
UCB$W_TT_PRTCTL	Port driver control flags. The bits in this field indicate features that are available to the port; the class driver specifies which of these features are to be enabled. The following fields are defined within UCB$W_TT_PRTCTL.	
	TTY$V_PC_NOTIME	No timeout. If set, the terminal class driver is not to set up timers for output.
	TTY$V_PC_DMAENA	DMA enabled. If set, DMA transfers are currently enabled on this port.
	TTY$V_PC_DMAAVL	DMA supported. If set, DMA transfers are supported for this port.
	TTY$V_PC_PRMMAP	Permanent map registers. If set, the port driver is to permanently allocate map registers.
	TTY$V_PC_MAPAVL	Map registers available. If set, the port driver has currently allocated map registers.
	TTY$V_PC_XOFAVL	Auto XOFF supported. If set, auto XOFF is supported for this port.
	TTY$V_PC_XOFENA	Auto XOFF enabled. If set, auto XOFF is currently enabled on this port.
	TTY$V_PC_NOCRLF	No auto line feed. If set, a line feed is not generated following a carriage return.

(continued on next page)

Table 10–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
	TTY$V_PC_BREAK	Break. If set, the port driver should generate break character; if clear, the port should turn off the break feature.
	TTY$V_PC_PORTFDT	FDT routine. If set, the port driver contains FDT routines.
	TTY$V_PC_NOMODEM	No modem. If set, the port cannot support modem operations.
	TTY$V_PC_NODISCONNECT	No disconnect. If set, the device cannot support virtual terminal operations.
	TTY$V_PC_SMART_READ	Smart read. If set, the port contains additional read capabilities.
	TTY$V_PC_ACCPORNAM	Access port name. If set, the port supports an access port name.
	TTY$V_PC_MULTISESSION	Multisession terminal. If set, the port is part of a multisession terminal.
UCB$L_TT_DS_ST	Current modem state.	
UCB$B_TT_DS_RCV	Current receive modem.	
UCB$B_TT_DS_TX	Current transmit modem.	
UCB$W_TT_DS_TIM	Current modem timeout.	
UCB$B_TT_MAINT	Maintenance functions. This field is used as the argument to the port driver's PORT_MAINT routine. It is written by the class driver and read by the port driver. It contains several bits that allow the following maintenance functions:	
	IO$M_LOOP	Set loopback mode.
	IO$M_UNLOOP	Reset loopback mode.

(continued on next page)

Table 10–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
	IO$M_AUTXOF_ENA	Enable the use of auto XON/XOFF on this line. This is the default.
	IO$M_AUTXOF_DIS	Disable the use of auto XON/XOFF on this line.
	IO$M_LINE_OFF	Disable interrupts on this line.
	IO$M_LINE_ON	Reenable interrupts on this line.
	Reference these bits by using the mask, shifted as follows:	

```
BITB   #IO$M_LOOP@-7,-
        UCB$B_TT_MAINT(R5);    Set loopback mode
```

Field	Use	
	UCB$B_TT_MAINT also defines the bit UCB$V_TT_DSBL that, when set, indicates that the line has been disabled.	
UCB$L_TT_FBK	Address of fallback block.	
UCB$L_TT_RDVERIFY	Address of read/verify table. Reserved for future use.	
UCB$L_TT_CLASS1	First class driver longword.	
UCB$L_TT_CLASS2	Second class driver longword.	
UCB$L_TT_ACCPORNAM	Address of counted string.	
UCB$L_TT_A_GCBADR	Glyph Control Block address	
UCB$W_TT_A_EDSTS	Multibyte line edit states	
UCB$B_TT_A_STATE	On-demand loading states	
UCB$B_TT_A_PARSE	ODL parse states	
UCB$B_TT_A_TRANS	JIS conversion states	
UCB$B_TT_A_XEDSTS	Extended line edit states	
UCB$L_TT_A_DECHSET	Default char set bitmask. The lowest byte of this field is known as UCB$B_TT_A_CHAR and represents the default Asian modes.	
UCB$L_TP_MAP	Map registers.	
UCB$B_TP_STAT	DMA port-specific status.	
	The following fields are defined within UCB$B_TP_STAT.	
	TTY$V_TP_ABORT	DMA abort requested on this line.
	TTY$V_TP_ALLOC	Allocate map fork in progress.
	TTY$V_TP_DLLOC	Deallocate map fork in progress.

10.15 VLE (Vector List Extension)

The driver loading mechanism (as directed by the SYSMAN command IO CONNECT) connects a hardware device to one or more interrupt vectors. Although most devices connected to VAX systems use preassigned vector locations, many devices on Alpha systems use 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 which contains a list of vectors for the device.

The vector list extension is described in Table 10–23.

Table 10–23 Contents of the Vector List Extension

Field	Use
VLE$PS_IDB	Address of the IDB with which the VLE is associated.
VLE$L_NUMVEC	Number of vector entries in the VLE.
VLE$W_SIZE	Size of VLE. The driver-loading procedure writes this field when it creates the VLE.
VLE$B_TYPE	Structure type. The driver loading procedure writes the constant DYN$C_MISC in this field.
VLE$B_SUBTYPE	Structure subtype. The driver loading procedure writes the constant DYN$C_VLE in this field.
VLE$L_VECTOR_LIST	Beginning of interrupt vector list. This field is an array of unsigned longwords containing the appropriate byte offset into either the SCB or the ADP vector table.

11

MACRO-32 Driver Macros

This chapter describes the JSB-replacement macros, FDT completion macros, and other macros used by OpenVMS Alpha device drivers.

Table 11–1 highlights some of the differences between OpenVMS VAX and OpenVMS Alpha macros.

Table 11–1 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
CPUDISP	Causes a branch to a specified address according to the CPU type of the Alpha 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

(continued on next page)

Table 11–1 (Cont.) New, Changed, and Unsupported OpenVMS Driver Macros

Macro	Description	Notes
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 $DRIVER_ERRRTN_ENTRY.	New
CRAM_ALLOC	Allocates a controller register access mailbox.	New
CRAM_CMD	Calculates the COMMAND, MASK, and RBADR fields for a hardware I/O mailbox according to the requirements of a specific I/O interconnect.	New
CRAM_DEALLOC	Deallocates a controller register access mailbox.	New
CRAM_IO	Queues the hardware I/O mailbox defined within a controller register access mailbox (CRAM) to the mailbox pointer register (MBPR) and awaits the completion of the mailbox transaction.	New
CRAM_QUEUE	Queues the hardware I/O mailbox defined within a controller register access mailbox (CRAM) to the mailbox pointer register (MBPR).	New
CRAM_WAIT	Awaits the completion of a hardware I/O mailbox transaction to a tightly coupled I/O interconnect.	New
DDTAB	Generates a driver dispatch table (DDT) labeled *devnam*$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

(continued on next page)

Table 11–1 (Cont.) New, Changed, and Unsupported OpenVMS Driver Macros

Macro	Description	Notes
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
$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

(continued on next page)

Table 11–1 (Cont.) New, Changed, and Unsupported OpenVMS Driver Macros

Macro	Description	Notes
$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 Alpha 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 11–1 (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 Alpha system executing the code in the macro expansion.	New
TBI_ALL	Invalidates the data and instruction translation buffers in their entirety.	New

(continued on next page)

Table 11–1 (Cont.) New, Changed, and Unsupported OpenVMS Driver Macros

Macro	Description	Notes
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
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

---

CALL_ABORTIO

Completes the servicing of an I/O request without returning status to the I/O status block specified in the request.

Format

CALL_ABORTIO [do_ret=YES]

Parameters

do_ret
Indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

Description

A JMP to EXE$ABORTIO in the FDT routine of a VAX driver should be replaced with the CALL_ABORTIO macro. It initializes the **irp**, **pcb**, **ucb**, and **qio_status** parameters from the contents of R3, R4, R5, and R0, respectively, and calls EXE_STD$ABORTIO. When EXE_STD$ABORTIO returns control to the code generated by a default invocation of CALL_ABORTIO, a RET instruction returns control to the caller of CALL_ABORTIO's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

CALL_ALLOCBUF, CALL_ALLOCIRP

Allocates a buffer from nonpaged pool for a buffered-I/O operation.

Format

CALL_ALLOCBUF

CALL_ALLOCIRP

Description

A JSB to EXE$ALLOCBUF and EXE$ALLOCIRP in a VAX driver should be replaced with CALL_ALLOCBUF and CALL_ALLOCIRP, respectively. CALL_ALLOCBUF calls EXE_STD$ALLOCBUF using the current contents of R1 as the **reqsize** argument. Both CALL_ALLOCBUF and CALL_ALLOCIRP return status in R0, the address of the allocated buffer in R2 and its size in R1. If a resource wait occurred, these macros return the address of the PCB in R4.

CALL_ALLOCEMB

Allocates an error message buffer and initializes its header.

Format

CALL_ALLOCEMB

Description

A JSB to ERL$ALLOCEMB in a VAX driver should be replaced with the CALL_ALLOCEMB macro. CALL_ALLOCEMB calls ERL_STD$ALLOCEMB using the current contents of R1 as the **size** argument. It returns status in R0, the address of the allocated EMB in R2 and copies the error log sequence number from EMB$W_DV_ERRSEQ to R1.

CALL_ALTQUEPKT

Delivers an IRP to a driver's alternate start-I/O routine without regard for the status of the device.

Format

CALL_ALTQUEPKT

Description

A JSB to EXE$ALTQUEPKT in a VAX driver should be replaced with the CALL_ALTQUEPKT macro. CALL_ALTQUEPKT calls EXE_STD$ALTQUEPKT, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

CALL_ALTREQCOM

Completes an I/O request for a device using the disk or tape class drivers.

Format

CALL_ALTREQCOM

Description

A JSB to IOC$ALTREQCOM in a VAX driver should be replaced with the CALL_ALTREQCOM macro. CALL_ALTREQCOM calls IOC_STD$ALTREQCOM, using the current contents of R0, R1, and R5 as the **iost1**, **iost2**, and **cdrp** arguments, respectively. When IOC_STD$ALTREQCOM returns, the macro returns the address of the IRP in R3 and the address of the UCB in R4.

CALL_BROADCAST

Broadcasts the specified message to a given terminal.

Format

CALL_BROADCAST [save_r1]

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to IOC_STD$BROADCAST. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

Description

A JSB to IOC$BROADCAST in a VAX driver should be replaced with the CALL_BROADCAST macro. CALL_BROADCAST calls IOC_STD$BROADCAST, using the current contents of R1, R2, and R5 as the **msglen**, **msg_p**, and **ucb** arguments, respectively. It returns status in R0. Unless you specify **save_r1**=**NO**, the macro preserves the quadword register R1 across the call.

CALL_CANCELIO

Conditionally marks a UCB so that its current I/O request will be canceled.

Format

CALL_CANCELIO [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to IOC_STD$CANCELIO. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not preserved.)

Description

A JSB to IOC$CANCELIO in a VAX driver should be replaced with the CALL_CANCELIO macro. CALL_CANCELIO calls IOC_STD$CANCELIO, using the current contents of R2, R3, R4, and R5 as the **chan**, **irp**, **pcb**, and **ucb** arguments, respectively. Unless you specify **save_r0r1**=**NO**, it preserves the quadword registers R0 and R1 across the call.

CALL_CARRIAGE

Interprets the carriage control specifier in IRP$B_CARCON and converts it to a generic prefix/suffix format.

Format

CALL_CARRIAGE

Description

A JSB to EXE$CARRIAGE in a VAX driver should be replaced with the CALL_CARRIAGE macro. CALL_CARRIAGE calls EXE_STD$CARRIAGE, using the current contents of R3 as the **irp** arguments.

CALL_CHK*xxx*ACCES

Checks logical (CALL_CHKLOGACCES), physical (CALL_CHKPHYACCES), read (CALL_CHKRDACCES), write (CALL_CHKWRTACCES), execute (CALL_CHKEXEACCES), create (CALL_CHKCREACCES), or delete (CALL_CHKDELACCES) I/O function access, based on the specified protection information.

Format

CALL_CHKCREACCES [save_r1]

CALL_CHKDELACCES [save_r1]

CALL_CHKEXEACCES [save_r1]

CALL_CHKLOGACCES [save_r1]

CALL_CHKPHYACCES [save_r1]

CALL_CHKRDACCES [save_r1]

CALL_CHKWRTACCES [save_r1]

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to EXE_STD$CHKPHYACCES, EXE_STD$CHKLOGACCES, EXE_STD$CHKWRTACCES, EXE_STD$CHKEXEACCES, EXE_STD$CHKCREACCES, EXE_STD$CHKDELACCES or EXE_STD$CHKRDACCES. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

Description

A JSB to EXE$CHKCREACCES, EXE$CHKDELACCES, EXE$CHKEXEACCES, EXE$CHKPHYACCES, EXE$CHKLOGA, EXE$CHKWRTACCES, or EXE$CHKRDACCES in a VAX driver should be replaced with the CALL_CHKCREACCES, CALL_CHKDELACCES, CALL_CHKEXEACCES, CALL_CHKLOGACCES, CALL_CHKPHYACCES, CALL_CHKWRTACCES, or CALL_CHKRDACCES macros respectively. Each macro calls the corresponding access-checking routine, using the current contents of R0, R1, R4, and R5 as the **arb**, **orb**, **pcb**, and **ucb** arguments. Unless you specify **save_r1**=**NO**, the macro preserves the quadword register R1 across the call. All macros return status in R0.

CALL_CLONE_UCB

Copies a template UCB and links it to the appropriate DDB list.

Format

CALL_CLONE_UCB [interface_warning=YES]

Parameters

[interface_warning=YES]
Specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the VAX version of the corresponding system routine. To suppress the warning, specify **interface_warning**=**NO**.

Description

A JSB to IOC$CLONE_UCB in a VAX driver should be replaced with CALL_CLONE_UCB. It calls IOC_STD$CLONE_UCB using the current contents of R5 as the **tmpl_ucb** argument. CALL_CLONE_UCB returns status in R0 and the address of the newly-created UCB in R2, but does not return the address of the UCBs that precede and follow it on the DDB chain in R3 and R1, respectively.

CALL_COPY_UCB

Copies and initializes a template UCB and ORB.

Format

CALL_COPY_UCB

Description

A JSB to IOC$COPY_UCB in a VAX driver should be replaced with the CALL_COPY_UCB macro. CALL_COPY_UCB calls IOC_STD$COPY_UCB using the current contents of R5 as the **src_ucb** argument. CALL_CLONEUCB returns the address of the newly-created UCB in R2.

CALL_CREDIT_UCB

Credits the UCB charges associated with a given UCB against the process identified by the contents of UCB$L_CPID.

Format

CALL_CREDIT_UCB

Description

A JSB to IOC$CREDIT_UCB in a VAX driver should be repleaced with CALL_CREDIT_UCB. CALL_CREDIT_UCB calls IOC_STD$CREDIT_UCB using the current contents of R5 as the **ucb** argument.

CALL_CVTLOGPHY

Conditionally converts a logical block number to a physical disk address and stores the result in the I/O request packet.

Format

CALL_CVTLOGPHY

Description

A JSB to IOC$CVTLOGPHY in a VAX driver should be replaced with the CALL_CVTLOGPHY macro. CALL_CVTLOGPHY calls IOC_STD$CVTLOGPHY, using the current contents of R0, R3, and R5 as the **lbn**, **irp** and **ucb** arguments, respectively.

CALL_CVT_DEVNAM

Converts a device name and unit number to a physical device name string.

Format

CALL_CVT_DEVNAM

Description

A JSB to IOC$CVT_DEVNAM in a VAX driver should be replaced with the CALL_CVT_DEVNAM macro. CALL_CVT_DEVNAM calls IOC_STD$CVT_DEVNAM, using the current contents of R0, R1, R4, and R5 as the **buflen**, **buf**, **form**, and **ucb** arguments, respectively.

The macro returns status in R0 and the length of the conversion string in R1.

CALL_DELATTNAST

Delivers all attention ASTs linked in the specified list.

Format

CALL_DELATTNAST [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to COM_STD$DELATTNAST. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

Description

A JSB to COM$DELATTNAST in a VAX driver should be replaced with the CALL_DELATTNAST macro. CALL_DELATTNAST calls COM_STD$DELATTNAST using the current contents of R4 and R5 as the **listhead** and **ucb** arguments, respectively. Unless you specify **save_r0r1**=**NO**, it preserves the quadword registers R0 and R1 across the call.

CALL_DELATTNASTP

Delivers all attention ASTs linked in the specified list for a given process.

Format

CALL_DELATTNASTP [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to COM_STD$DELATTNASTP. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

Description

A JSB to COM$DELATTNASTP in a VAX driver should be replaced with the CALL_DELATTNASTP macro. CALL_DELATTNASTP calls COM_STD$DELATTNASTP using the current contents of R4, R5 and R6 as the **listhead**, **ucb**, and **ipid** arguments, respectively. Unless you specify **save_r0r1**=**NO**, the macro preserves the quadword registers R0 and R1 across the call.

CALL_DELCTRLAST

Delivers all control ASTs, linked in the specified list, that match a given condition.

Format

CALL_DELCTRLAST [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to COM_STD$DELCTRLAST. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

Description

A JSB to COM$DELCTRLAST in a VAX driver should be replaced with the CALL_DELCTRLAST macro. CALL_DELCTRLAST calls COM_STD$DELCTRLAST using the current contents of R4, R5, and R3 as the **listhead**, **ucb**, and **matchchar** arguments, respectively. When COM$DELCTRLAST returns, it moves the include character into R3. Unless you specify **save_r0r1**=**NO**, it preserves the quadword registers R0 and R1 across the call.

CALL_DELCTRLASTP

Delivers all control ASTs, linked in the specified list, that match a given condition.

Format

CALL_DELCTRLASTP [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to COM_STD$DELCTRLASTP. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

Description

A JSB to COM$DELCTRLASTP in a VAX driver should be replaced with the CALL_DELCTRLASTP macro. CALL_DELCTRLASTP calls COM_STD$DELCTRLASTP using the current contents of R4, R5, R6, and R3 as the **listhead**, **ucb**, **ipid**, and **matchchar** arguments, respectively. When COM$DELCTRLASTP returns, it moves the include character into R3. Unless you specify **save_r0r1**=**NO**, it preserves the quadword registers R0 and R1 across the call.

CALL_DELETE_UCB

Deletes the specified UCB if its reference count is zero and UCB$V_DELETEUCB is set in UCB$L_STS.

Format

CALL_DELETE_UCB

Description

A JSB to IOC$DELETE_UCB in a VAX driver should be replaced with the CALL_DELETE_UCB macro. CALL_DELETE_UCB calls IOC_STD$DELETE_UCB using the current contents of R5 as the **ucb** argument.

CALL_DEVICEATTN, CALL_DEVICERR, CALL_DEVICTMO

Allocate an error message buffer and record in it information concerning the error.

Format

CALL_DEVICEATTN [save_r0r1]

CALL_DEVICERR [save_r0r1]

CALL_DEVICTMO [save_r0r1]

Parameters

save_r0r1
Indicates that the macros must preserve the contents of R0 and R1 across the call to ERL_STD$DEVICEATTN, ERL_STD$DEVICERR, or ERL_STD$DEVICTMO. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

Description

JSBs to ERL$DEVICEATTN, ERL$DEVICERR, and ERL$DEVICTMO in a VAX driver should be replaced with the CALL_DEVICEATTN, CALL_DEVICERR, and CALL_DEVICTMO macros, respectively. Each macro calls the corresponding routine using the current contents of R4 and R5 as the **driver_param** and **ucb** arguments, respectively. Unless you specify **save_r0r1**=**NO**, it preserves the quadword registers R0 and R1 across the call.

CALL_DIAGBUFILL

Fills a diagnostic buffer if the original $QIO request specified such a buffer.

Format

CALL_DIAGBUFILL

Description

A JSB to IOC$DIAGBUFILL in a VAX driver should be replaced with the CALL_DIAGBUFILL macro. CALL_DIAGBUFILL calls IOC_STD$DIAGBUFILL, using the current contents of R4 and R5 as the **driver_param** and **ucb** arguments, respectively.

CALL_DRVDEALMEM

Deallocates system dynamic memory.

Format

CALL_DRVDEALMEM   [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to COM_STD$DRVDEALMEM. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

Description

A JSB to COM$DRVDEALMEM in a VAX driver should be replaced with the CALL_DRVDEALMEM macro. CALL_DRVDEALMEM calls COM_STD$DRVDEALMEM using the current contents of R0 as the **ptr** argument. Unless you specify **save_r0r1**=**NO**, the macro preserves the quadword registers R0 and R1 across the call.

CALL_FILSPT

Fills a system page-table entry (PTE) with the transfer PTE of a buffer that is locked in memory so that the system PTE may be directly addressed.

Format

CALL_FILSPT

Description

A JSB to IOC$FILSPT in a VAX driver should be replaced with the CALL_FILSPT macro. CALL_FILSPT calls IOC_STD$FILSPT, passing the current contents of R5 as the **ucb** argument. It returns in R0 the system virtual address of the first byte in the page that contains the buffer.

CALL_FINISHIO, CALL_FINISHIOC, CALL_FINISHIO_NOIOST

Complete the servicing of an I/O request and return status to the I/O status block specified in the original call to the $QIO system service.

Format

CALL_FINISHIO [do_ret=YES]

CALL_FINISHIOC [do_ret=YES]

CALL_FINISHIO_NOIOST [do_ret=YES]

Parameters

do_ret
Indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

Description

JMPs to EXE$FINISHIO, EXE$FINISHIOC, and EXE$FINISHIO_NOIOST in a VAX driver should be replaced with the CALL_FINISHIO, CALL_FINISHIOC, and CALL_FINISHIO_NOIOST macros, respectively. CALL_FINISHIO moves the current contents of R0 and R1 into IRP$L_IOST1 and IRP$L_IOST2, respectively; CALL_FINISHIOC initializes IRP$L_IOST1 from R0 and clears IRP$L_IOST2; and CALL_FINISHIO_NOIOST fills in neither IRP field. The macros initialize the **irp** and **ucb** parameters from the contents of R3 and R5, respectively before calling EXE_STD$FINISHIO. When EXE_STD$FINISHIO returns control to the code generated by a default invocation of CALL_FINISHIO, CALL_FINISHIOC, or CALL_FINISHIO_NOIOST, a RET instruction returns control to the caller of the macro's invoker.

Status is returned in R0 and in the FDT_CONTEXT structure.

CALL_FLUSHATTNS

Removes specified ASTs from an attention AST list.

Format

CALL_FLUSHATTNS

Description

A JSB to COM$FLUSHATTNS in a VAX driver should be replaced with the CALL_FLUSHATTNS macro. CALL_FLUSHATTNS calls COM_STD$FLUSHATTNS using the current contents of R4, R5, R6, and R7 as the **pcb**, **ucb**, **chan**, and **acb_lh** arguments, respectively. It returns status in R0.

CALL_FLUSHCTRLS

Removes specified ASTs from a control AST list.

Format

CALL_FLUSHCTRLS

Description

A JSB to COM$FLUSHCTRLS in a VAX driver should be replaced with the CALL_FLUSHCTRLS macro. CALL_FLUSHCTRLS calls COM_STD$FLUSHCTRLS using the current contents of R2, R4, R5, R6, and R7 as the **mask**, **pcb**, **ucb**, **chan**, and **acb_lh** arguments, respectively. It returns status in R0.

CALL_GETBYTE

Fetches a single byte of data from a user buffer.

Format

CALL_GETBYTE

Description

A JSB to IOC$GETBYTE in a VAX driver should be replaced with the CALL_GETBYTE macro. CALL_GETBYTE calls IOC_STD$GETBYTE, passing the current contents of R0 and R5 as the **sva** and **ucb** arguments, respectively. It returns in R0 the byte of data (not zero-extended) returned from the user buffer. It returns in R1 the updated system virtual address. (Note that this differs from the behavior of IOC$GETBYTE, which returns the byte of data in R1 and the updated system virtual address in R0.)

CALL_INITBUFWIND

Initializes a single-page window into a user buffer.

Format

CALL_INITBUFWIND

Description

A JSB to IOC$INITBUFWIND in a VAX driver should be replaced with the CALL_INITBUFWIND macro. CALL_INITBUFWIND calls IOC_STD$INITBUFWIND, passing the current contents of R5 as the **ucb** argument. It returns in R0 the system virtual address of the first byte in the page that contains the buffer.

CALL_INITIATE

Initiates the processing of the next I/O request for a device unit.

Format

CALL_INITIATE

Description

A JSB to IOC$INITIATE in a VAX driver should be replaced with the CALL_INITIATE macro. CALL_INITIATE calls IOC_STD$INITIATE, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

CALL_INSERT_IRP

Inserts an I/O request packet (IRP) into the specified queue of IRPs according to the base priority of the process that issued the I/O request.

Format

CALL_INSERT_IRP

Description

A JSB to EXE$INSERT_IRP in a VAX driver should be replaced with the CALL_INSERT_IRP macro. CALL_INSERT_IRP calls EXE_STD$INSERT_IRP, using the current contents of R2 and R3 as the **irp_lh** and **irp** arguments, respectively. It returns status in R0.

CALL_IOLOCK

Locks process pages in memory.

Format

CALL_IOLOCK

Description

A JSB to MMG$IOLOCK in a VAX driver should be replaced with the CALL_IOLOCK macro. CALL_IOLOCK calls MMG_STD$IOLOCK using the current contents of R0, R1, R2, and R4 as the **buf**, **bufsize**, **is_read**, and **pcb** arguments, respectively.

CALL_IOLOCK returns status in R0. If R0 contains SS$_NORMAL, R1 contains the system virtual address of the first page-table entry. If R0 contains zero, R1 contains the address of a page to be faulted into memory. R0 can also contain a system-level status.

CALL_IOLOCKR

Locks the I/O database mutex on behalf of its caller for read access.

Format

CALL_IOLOCKR save_r1

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to SCH_STD$IOLOCKR. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

Description

A JSB to SCH$IOLOCKR in a VAX driver should be replaced with the CALL_IOLOCKR macro. CALL_IOLOCKR calls SCH_STD$IOLOCKR using the current contents of R4 as the **pcb** argument.

CALL_IOLOCKR returns the address of the I/O database mutex in R0. Unless you specify **save_r1**=**NO**, the macro preserves R1 across the call.

CALL_IOLOCKW

Locks the I/O database mutex on behalf of its caller for write access.

Format

CALL_IOLOCKW save_r1

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to SCH_STD$IOLOCKW. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

Description

A JSB to SCH$IOLOCKW in a VAX driver should be replaced with the CALL_IOLOCKW macro. CALL_IOLOCKW calls SCH_STD$IOLOCKW using the current contents of R4 as the **pcb** argument.

CALL_IOLOCKW returns the address of the I/O database mutex in R0. Unless you specify **save_r1**=**NO**, the macro preserves R1 across the call.

CALL_IORSNWAIT

Places a process in a resource wait state if it has enabled resource waits.

Format

CALL_IORSNWAIT [do_ret=YES]

Parameters

do_ret
Indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

Description

A JMP to EXE$IORSNWAIT in a VAX driver should be replaced with the CALL_IORSNWAIT macro. CALL_IORSNWAIT calls EXE_STD$IORSNWAIT using the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **qio_status**, and **rsn** arguments, respectively. When EXE_STD$IORSNWAIT returns control to the code generated by a default invocation of $IORSNWAIT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

CALL_IOUNLOCK

Releases ownership of the I/O database mutex and, if the mutex has thus become available to waiting processes, reactivates the next eligible process.

Format

CALL_IOUNLOCK

Description

A JSB to SCH$IOUNLOCK in a VAX driver should be replaced with the CALL_IOUNLOCK macro. CALL_IOUNLOCK calls SCH_STD$IOUNLOCK using the current contents of R4 as the **pcb** argument.

CALL_LINK_UCB

Searches the UCB list attached to the device data block identified by the specified UCB and links the specified UCB into the list in ascending unit number order.

Format

CALL_LINK_UCB [interface_warning=YES]

Parameters

[interface_warning=YES]
Specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the VAX version of the corresponding system routine. **interface_warning**=**NO** suppresses the warning.

Description

A JSB to IOC$LINK_UCB in a VAX driver should be replaced with the CALL_LINK_UCB macro. CALL_LINK_UCB calls IOC_STD$LINK_UCB using the current contents of R5 as the **ucb** argument. CALL_LINK_UCB returns the status in R0 and address of the newly-created UCB in R2, but does not return the address of the UCBs that precede and follow it on the DDB chain in R3 and R1, respectively.

CALL_MAPVBLK

Maps a virtual block to a logical block using a mapping window.

Format

CALL_MAPVBLK

Description

A JSB to IOC$MAPVBLK in a VAX driver should be replaced with the CALL_MAPVBLK macro. CALL_MAPVBLK calls IOC_STD$MAPVBLK, using the current contents of R0, R1, R2, R3, and R5 as the **vbn**, **numbytes**, **wcb**, **irp** and **ucb** arguments, respectively. It returns status in R0, the address of the logical block number of the first block mapped in R1, the number of unmapped bytes in R2, and the address of the updated UCB in R3. If the low bit of the status value in R0 is clear, signifying failure status, only the value in R2 is valid.

CALL_MNTVER

Assists a driver with mount verification.

Format

CALL_MNTVER

Description

A JSB to IOC$MNTVER in a VAX driver should be replaced with the CALL_MNTVER macro. CALL_MNTVER calls IOC_STD$MNTVER, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively.

CALL_MNTVERSIO

Processes I/O functions that affect the online count and local valid status of a disk.

Format

CALL_MNTVERSIO

Description

A JSB to EXE$MNTVERSIO in a VAX driver should be replaced with the CALL_MNTVERSIO macro. CALL_MNTVERSIO calls EXE_STD$MNTVERSIO, using the current contents of R0, R3, and R5 as the **rout**, **irp**, and **ucb** arguments, respectively.

CALL_MODIFYLOCK, CALL_MODIFYLOCK_ERR

Validate and prepare a user buffer for a direct-I/O, DMA read/write operation.

Format

CALL_MODIFYLOCK

CALL_MODIFYLOCK_ERR [interface_warning=YES]

Parameters

[interface_warning=YES]
Specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the VAX version of the corresponding system routine. **interface_warning=NO** suppresses the warning.

Description

A JSB to EXE$MODIFYLOCK in a VAX driver should be replaced with the CALL_MODIFYLOCK macro. A JSB to EXE$MODIFYLOCK_ERR should be replaced with the CALL_MODIFYLOCK_ERR macro. CALL_MODIFYLOCK calls EXE_STD$MODIFYLOCK, specifying 0 as the **err_rout** argument; CALL_MODIFYLOCK_ERR also calls EXE_STD$MODIFYLOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsize** arguments, respectively.

When EXE_STD$MODIFYLOCK or EXE_STD$MODIFYLOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro moves the contents of IRP$L_SVAPTE into R1 and writes a 5 into R2 to indicate a modify operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro writes a 5 to R2 to indicate a modify operation and and returns to FDT dispatching code in the $QIO system service.

CALL_MOUNT_VER

During I/O postprocessing, determines whether mount verification should be initiated on a given disk or tape device on behalf of the I/O request being completed.

Format

CALL_MOUNT_VER   [save_r0r1]

Parameters

save_r0r1
Indicates that the macro must preserve the contents of R0 and R1 across the call to EXE_STD$MOUNT_VER. If **save_r0r1** is blank or **save_r0r1**=**YES**, the 64-bit registers are saved. (In the former case, the macro generates a compile-time message. If **save_r0r1**=**NO**, the registers are not saved.)

Description

A JSB to EXE$MOUNT_VER in a VAX driver should be replaced with the CALL_MOUNT_VER macro. CALL_MOUNT_VER calls EXE_STD$MOUNT_VER, using the current contents of R0, R1, R3, and R5 as the **iost1**, **iost2**, **irp**, and **ucb** arguments, respectively. When EXE_STD$MOUNT_VER returns, code generated by this macro copies return status from R0 to R2. Unless you specify **save_r0r1**=**NO**, the macro preserves the quadword registers R0 and R1 across the call.

CALL_MOVFRUSER, CALL_MOVFRUSER2

Move data from a user buffer to a device.

Format

CALL_MOVFRUSER

CALL_MOVFRUSER2

Description

JSBs to IOC$MOVFRUSER and IOC$MOVFRUSER2 in a VAX driver should be replaced with CALL_MOVFRUSER and CALL_MOVFRUSER2, respectively. CALL_MOVFRUSER calls IOC_STD$MOVFRUSER, and CALL_MOVFRUSER2 calls IOC_STD$MOVFRUSER2, passing the current contents of R1, R2, and R5 as the **sysbuf**, **numbytes**, and **ucb** arguments. CALL_MOVFRUSER2 also passes the current contents of R0 as the **sva** argument. Both macros return in R0 and R1, respectively, the system virtual addresses of the bytes in the internal buffer and user buffer after the last byte moved.

CALL_MOVTOUSER, CALL_MOVTOUSER2

Move data from an internal buffer to a user buffer.

Format

CALL_MOVTOUSER

CALL_MOVTOUSER2

Description

JSBs to IOC$MOVTOUSER and IOC$MOVTOUSER2 in a VAX driver should be replaced with CALL_MOVTOUSER and CALL_MOVTOUSER2, respectively. CALL_MOVTOUSER calls IOC_STD$MOVTOUSER, and CALL_MOVTOUSER2 calls IOC_STD$MOVTOUSER2, passing the current contents of R1, R2, and R5 as the **sysbuf**, **numbytes**, and **ucb** arguments. CALL_MOVTOUSER2 also passes the current contents of R0 as the **sva** argument. Both macros return in R0 and R1, respectively, the system virtual addresses of the bytes in the internal buffer and user buffer after the last byte moved.

CALL_PARSDEVNAM

Parses a device name string, checking its syntax and extracting the node name, allocation class number, and unit number.

Format

CALL_PARSDEVNAM

Description

A JSB to IOC$PARSDEVNAM in a VAX driver should be replaced with the CALL_PARSDEVNAM macro. CALL_PARSDEVNAM calls IOC_STD$PARSDEVNAM, using the current contents of R8, R9, and R10 as the **devnamsiz**, **devnam**, and **flags** arguments, respectively. When IOC_STD$PARSDEVNAM returns, the macro returns status in R0; the unit number in R2; the length of the SCS node name at the beginning of the name string, allocation class number, or device type code in R3; the size of the name string in R8, the address of the name string in R9, and the flags in R10.

CALL_POST, CALL_POST_NOCNT

Initiate device-independent postprocessing of an I/O request independent of the status of the device unit.

Format

CALL_POST [save_r1]

CALL_POST_NOCNT [save_r1]

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to COM_STD$POST or COM_STD$POST_NOCNT. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

Description

A JSB to COM$POST in a VAX driver should be replaced with the CALL_POST macro. CALL_POST calls COM_STD$POST using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively. CALL_POST_NOCNT calls COM_STD$POST_NOCNT using the current contents of R3 as the **irp** argument. Unless you specify **save_r1**=**NO**, the macro preserves the quadword register R1 across the call.

CALL_POST_IRP

Inserts an I/O request packet in a CPU-specific I/O postprocessing queue.

Format

CALL_POST_IRP

Description

A JSB to IOC$POST_IRP in a VAX driver should be replaced with the CALL_POST_IRP macro. CALL_POST_IRP calls IOC_STD$POST_IRP using the current contents of R3 as the **irp** argument.

CALL_PTETOPFN

Returns a page frame number (PFN) from a page-table entry (PTE) that has already been determined to be invalid.

Format

CALL_PTETOPFN

Description

A JSB to IOC$PTETOPFN in a VAX driver should be replaced with the CALL_PTETOPFN macro. CALL_PTETOPFN extracts the quadword page-table entry from R3 and passes a pointer to it as the **pte** argument to IOC_STD$PTETOPFN. It returns the page frame number in R0.

CALL_QIOACPPKT

Delivers an IRP to the appropriate ACP or XQP.

Format

CALL_QIOACPPKT [do_ret=YES]

Parameters

do_ret
Indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

Description

A JMP to EXE$QIOACPPKT in a VAX driver should be replaced with the CALL_QIOACPPKT macro. CALL_QIOACPPKT calls EXE_STD$QIOACPPKT using the current contents of R3, R4, and R5 as the **irp**, **pcb**, and **ucb** arguments, respectively. When EXE_STD$QIOACPPKT returns control to the code generated by a default invocation of $QIOACPPKT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0 and in the FDT_CONTEXT structure.

CALL_QIODRVPKT

Delivers an IRP to the driver's start-I/O routine or pending-I/O queue.

Format

CALL_QIODRVPKT [do_ret=YES]

Parameters

do_ret
Indicates that the macro generates a RET instruction at the end of its expansion, thus returning control to the caller of the routine that invokes it.

Description

A JMP to EXE$QIODRVPKT in a VAX driver should be replaced with the CALL_QIODRVPKT macro. CALL_QIODRVPKT clears IRP$PS_FDT_CONTEXT and calls EXE_STD$INSIOQ, using the current contents of R3 and R5 as the **irp** and **ucb** arguments, respectively. When EXE_STD$INSIOQ returns control to the code generated by a default invocation of CALL_QIODRVPKT, a RET instruction returns control to the caller of the macro's invoker. Status is returned in R0.

CALL_QNXTSEG1

Queues the next segment of a virtual I/O request that did not map to a single contiguous I/O request.

Format

CALL_QNXTSEG1

Description

A JSB to IOC$QNXTSEG1 in a VAX driver should be replaced with the CALL_QNXTSEG1 macro. CALL_QNXTSEG1 calls IOC_STD$QNXTSEG1 using the current contents of R0, R1, R2, R3, R4, and R5 as the **vbn**, **bcnt**, **wcb**, **irp**, **pcb**, and **ucb** arguments. It returns the address of the updated UCB in R5.

CALL_QXQPPKT

Inserts an IRP on the end of the XQP work queue and initiates its processing if it is the only request on the queue.

Format

CALL_QXQPPKT

Description

A JMP to EXE$QXQPPKT in a VAX driver should be replaced with the CALL_QXQPPKT macro. CALL_QXQPPKT calls EXE_STD$QXQPPKT using the current contents of R4 and R5 as the **pcb** and **acb** arguments, respectively. Status is returned in R0 and in the FDT_CONTEXT structure.

CALL_READCHK, CALL_READCHKR

Verifies that a process has write access to the pages in the buffer specified in a $QIO request.

Format

CALL_READCHK

CALL_READCHKR

Description

A JSB to EXE$READCHK in a VAX driver should be replaced with the CALL_READCHK macro. A JSB to EXE$READCHKR should be replaced with the CALL_READCHKR macro. Both macros call EXE_STD$READCHK using the current contents of R3, R4, R5, R0, and R1 as the **irp**, **pcb**, **ucb**, **buf**, and **bufsize** arguments, respectively.

When EXE_STD$READCHK returns, CALL_READCHK and CALL_READCHKR move 1 into R2 to indicate a read operation and examines the return status:

- If success status (SS$_NORMAL) is returned, CALL_READCHK and CALL_READCHKR copy the contents of IRP$L_BCNT into R1. CALL_READCHK writes the starting address of the I/O buffer in R0; CALL_READCHKR preserves the return status value in R0.
- If failure status (SS$_FDT_COMPL) is returned, CALL_READCHK returns to FDT dispatching code in the $QIO system service. CALL_READCHKR does not return control to $QIO.

CALL_READLOCK, CALL_READLOCK_ERR

Validate and prepare a user buffer for a direct-I/O, DMA write operation.

Format

CALL_READLOCK

CALL_READLOCK_ERR [interface_warning=YES]

Parameters

[interface_warning=YES]
Specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the VAX version of the corresponding system routine. **interface_warning**=**NO** suppresses the warning.

Description

A JSB to EXE$READLOCK in a VAX driver should be replaced with the CALL_READLOCK macro. A JSB to EXE$READLOCK_ERR in a VAX driver should be replaced with CALL_READLOCK_ERR. CALL_READLOCK calls EXE_STD$READLOCK, specifying 0 as the **err_rout** argument; CALL_READLOCK_ERR also calls EXE_STD$READLOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsize** arguments, respectively.

When EXE_STD$READLOCK or EXE_STD$READLOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro copies the contents of IRP$L_SVAPTE into R1 and writes a 1 to R2 to indicate a read operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro writes a 1 to R2 to indicate a read operation and returns to FDT dispatching code in the $QIO system service.

CALL_RELCHAN

Releases device ownership of all controller data channels.

Format

CALL_RELCHAN

Description

A JSB to IOC$RELCHAN in a VAX driver should be replaced with the CALL_RELCHAN macro. CALL_RELCHAN calls IOC_STD$RELCHAN using the current contents of R5 as the **ucb** argument.

CALL_RELEASEMB

Releases an error message buffer to the error-logging process.

Format

CALL_RELEASEMB

Description

A JSB to ERL$RELEASEMB in a VAX driver should be replaced with the CALL_RELEASEMB macro. CALL_RELEASEMB calls ERL_STD$RELEASEMB using the current contents of R2 as the **embdv** argument.

CALL_REQCOM

Completes an I/O operation on a device unit, requests I/O postprocessing of the current request, and starts the next I/O request waiting for the device.

Format

CALL_REQCOM

Description

A JSB to IOC$REQCOM in a VAX driver should be replaced with the CALL_REQCOM macro. CALL_REQCOM calls IOC_STD$REQCOM, using the current contents of R0, R1, and R5 as the **iost1**, **iost2**, and **ucb** arguments, respectively.

CALL_SEARCHDEV

Searches the I/O database for a specific physical device.

Format

CALL_SEARCHDEV

Description

A JSB to IOC$SEARCHDEV in a VAX driver should be replaced with the CALL_SEARCHDEV macro. CALL_SEARCHDEV calls IOC_STD$SEARCHDEV, using the current contents of R1 as the **descr_p** argument. When IOC_STD$SEARCHDEV returns, the macro returns returns status in R0, the UCB address in R1, the DDB address in R2, and the SB address in R3.

CALL_SEARCHINT

Searches the I/O database for the specified device, using specified search rules.

Format

CALL_SEARCHINT

Description

A JSB to IOC$SEARCHINT in a VAX driver should be replaced with the the CALL_SEARCHINT macro. CALL_SEARCHINT calls IOC_STD$SEARCHINT, using the current contents of R2, R3, R8, R9 and R10 as the **unit**, **scslen**, **devnamlen**, **devnam**, and **flags** arguments, respectively. When IOC_STD$SEARCHINT returns, the macro returns status in R0, the UCB address in R5, the DDB address in R6, and the SB address in R7.

CALL_SETATTNAST

Enables or disables attention ASTs.

Format

CALL_SETATTNAST

Description

A JSB to COM$SETATTNAST in a VAX driver should be replaced with the CALL_SETATTNAST macro. CALL_SETATTNAST calls COM_STD$SETATTNAST using the current contents of R3, R4, R5, R6, and R7, as the **irp**, **pcb**, **ucb**, **ccb**, and **acb_lh** arguments, respectively. It returns status in R0 and in the FDT_CONTEXT structure.

CALL_SETCTRLAST

Enables or disables control ASTs.

Format

CALL_SETCTRLAST

Description

A JSB to COM$SETCTRLAST in a VAX driver should be replaced with the CALL_SETCTRLAST macro. CALL_SETCTRLAST calls COM_STD$SETCTRLAST using the current contents of R3, R4, R5, R7, and R2, as the **irp**, **pcb**, **ucb**, **acb_lh**, and **mask** arguments, respectively. It returns the TAST block in R2. It returns status in R0 and in the FDT_CONTEXT structure.

CALL_SEVER_UCB

Removes the specified UCB from the UCB list of the device data block identified within the specified UCB.

Format

CALL_SEVER_UCB

Description

A JSB to IOC$SEVER_UCB in a VAX driver should be replaced with the CALL_SEVER_UCB macro. CALL_SEVER_UCB calls IOC_STD$SEVER_UCB using the current contents of R5 as the **ucb** argument.

CALL_SIMREQCOM

Completes an I/O operation by setting an event flag, modifying an I/O status block (IOSB), setting an event flag, or queuing an AST to the process requesting the I/O. The caller of this routine is responsible for checking quotas and updating the I/O count.

Format

CALL_SIMREQCOM

Description

A JSB to IOC$SIMREQCOM in a VAX driver should be replaced with the CALL_SIMREQCOM macro. CALL_SIMREQCOM calls IOC_STD$SIMREQCOM, using the current contents of R1, R2, R3, R4, R5, and R6 as the **iosb**, **pri**, **efn**, **iost**, **acb**, and **acmode** arguments, respectively.

CALL_SNDEVMSG

Builds and sends a device-specific message to the mailbox of a system process, such as the job controller or OPCOM.

Format

CALL_SNDEVMSG [save_r1]

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to COM_STD$POST. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

Description

A JSB to EXE$SNDEVMSG in a VAX driver should be replaced with the the CALL_SNDEVMSG macro. CALL_SNDEVMSG calls EXE_STD$SNDEVMSG, using the current contents of R3, R4, and R5 as the **mb_ucb**, **msgtyp**, and **ucb** arguments, respectively. It returns status in R0. Unless you specify **save_r1**=**NO**, the macro preserves the R1 across the call.

CALL_THREADCRB

Threads a controller request block (CRB) onto the due-time chain headed by IOC$GL_CRBTMOUT.

Format

CALL_THREADCRB [save_r0]

Parameters

save_r0
Indicates that the macro must preserve the contents of R0 across the call to IOC_STD$THREADCRB. If **save_r0** is blank or **save_r0**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r0**=**NO**, R0 is not saved.)

Description

A JSB to IOC$THREADCRB in a VAX driver should be replaced with the CALL_THREADCRB macro. CALL_THREADCRB calls IOC_STD$THREADCRB using the current contents of R3 as the **crb** argument. Unless you specify **save_r1**=**NO**, the macro preserves the quadword register R1 across the call.

CALL_UNLOCK

Unlocks process pages previously locked for a direct-I/O operation.

Format

CALL_UNLOCK

Description

A JSB to MMG$UNLOCK in a VAX driver should be replaced with the CALL_UNLOCK macro. CALL_UNLOCK calls MMG_STD$UNLOCK using the current contents of R1 and R3 as the **npages** and **svapte** arguments, respectively.

CALL_WRITECHK, CALL_WRITECHKR

Verify that a process has read access to the pages in the buffer specified in a $QIO request.

Format

CALL_WRITECHK

CALL_WRITECHKR

Description

A JSB to EXE$WRITECHK in a VAX driver should be replaced with the CALL_WRITECHK macro. A JSB to EXE$READCHKR in a VAX driver should be replaced with the CALL_READCHKR macro. Both macros call EXE_STD$READCHK using the current contents of R3, R4, R5, R0, and R1 as the **irp**, **pcb**, **ucb**, **buf**, and **bufsize** arguments, respectively.

When EXE_STD$WRITECHK returns, CALL_WRITECHK and CALL_WRITECHKR clear R2 to indicate a write operation and examines the return status:

- If success status (SS$_NORMAL) is returned, CALL_WRITECHK and CALL_WRITECHKR copy the contents of IRP$L_BCNT into R1. CALL_WRITECHK writes the starting address of the I/O buffer in R0; CALL_WRITECHKR preserves the return status value in R0.
- If failure status (SS$_FDT_COMPL) is returned, CALL_WRITECHK returns to FDT dispatching code in the $QIO system service. CALL_WRITECHKR does not return control to $QIO.

CALL_WRITELOCK, CALL_WRITELOCK_ERR

Validate and prepare a user buffer for a direct-I/O, DMA read operation.

Format

CALL_WRITELOCK

CALL_WRITELOCK_ERR [interface_warning=YES]

Parameters

[interface_warning=YES]
Specifies that the macro generate a compile-time warning indicating how the behavior of the macro differs from the VAX version of the corresponding system routine. **interface_warning**=**NO** suppresses the warning.

Description

A JSB to EXE$WRITELOCK in a VAX driver should be replaced with the CALL_WRITELOCK macro. A JSB to EXE$WRITELOCK_ERR in a VAX driver should be replaced with the CALL_WRITELOCK_ERR macro. CALL_WRITELOCK calls EXE_STD$WRITELOCK, specifying 0 as the **err_rout** argument; CALL_WRITELOCK_ERR also calls EXE_STD$WRITELOCK, using the contents of R2 as the **err_rout** argument. Both macros supply the current contents of R3, R4, R5, R6, R0, and R1 as the **irp**, **pcb**, **ucb**, **ccb**, **buf**, and **bufsize** arguments, respectively.

When EXE_STD$WRITELOCK or EXE_STD$WRITELOCK_ERR returns, code generated by the macro examines the return status:

- If success status (SS$_NORMAL) is returned, the macro moves the contents of IRP$L_SVAPTE into R1 and clears R2 to indicate a write operation. Status is returned in R0 and in the FDT_CONTEXT structure.
- If failure status (SS$_FDT_COMPL) is returned, the macro clears R2 to indicate a write operation and returns to FDT dispatching code in the $QIO system service.

CALL_WRTMAILBOX

Sends a message to a mailbox.

Format

CALL_WRTMAILBOX [save_r1]

Parameters

save_r1
Indicates that the macro must preserve the contents of R1 across the call to COM_STD$POST. If **save_r1** is blank or **save_r1**=**YES**, the 64-bit register is saved. (In the former case, the macro generates a compile-time message. If **save_r1**=**NO**, R1 is not saved.)

Description

A JSB to EXE$WRTMAILBOX in a VAX driver should be replaced with the CALL_WRTMAILBOX macro. CALL_WRTMAILBOX calls EXE_STD$WRTMAILBOX, using the current contents of R5, R3, and R4 as the **mb_ucb**, **msgsiz**, and **msg** arguments, respectively. It returns status in R0. Unless you specify **save_r1**=**NO**, the macro preserves the R1 across the call.

CLASS_UNIT_INIT

Generates the common code that must be executed by the unit initialization routine of all terminal port drivers.

Format

CLASS_UNIT_INIT [ucb=R5] [,port_vector=R0]

Parameters

[ucb=R5]
Address of UCB.

[port_vector=R0]
Address of port driver vector table.

Description

A terminal port driver's unit initialization routine invokes the CLASS_UNIT_INIT macro to perform initialization tasks common to all port drivers. To use the CLASS_UNIT_INIT macro, the driver must include an invocation of the $TTYMACS definition macro (from SYS$LIBRARY:LIB.MLB).

The CLASS_UNIT_INIT macro binds the terminal port and class driver into a single, complete driver by initializing the following fields as indicated:

Field	Contents
UCB$L_TT_CLASS	Class driver vector table address
UCB$L_TT_PORT	Port driver vector table address
UCB$L_TT_GETNXT	Procedure value of the class driver's get-next-character routine (CLASS_GETNXT)
UCB$L_TT_PUTNXT	Procedure value of the class driver's put-next-character routine (CLASS_PUTNXT)
UCB$B_TT_PARITY	Current parity, frame, and stop bit information (from TTY$GB_PARITY)
UCB$B_TT_DEPARI	Default parity, frame, and stop bit information (from TTY$GB_PARITY)
DDT$PS_START	Procedure value of the class driver's start-I/O routine
DDT$PS_FDT	Address of the class driver's function-decision table
DDT$PS_CANCEL	Procedure value of the class driver's cancel-I/O routine
DDT$PS_ALTSTART	Procedure value of the class driver's alternate start-I/O routine

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

Because an OpenVMS Alpha terminal port driver cannot share a single DDT with the OpenVMS Alpha terminal class driver, the CLASS_UNIT_INIT macro does not write the address of the class_driver's DDT into UCB$L_DDT. Rather, it assumes that the port driver has created its own DDT with entries for its controller initialization routine (DDT$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.

CPUDISP

Causes a branch to a specified address according to the CPU type of the Alpha processor executing the code generated by the macro expansion.

Format

CPUDISP list [,continue=YES]

Parameters

list

List containing one or more pairs of arguments in the following format:

<**CPU-type**, **destination**>

The **CPU-type** parameter identifies the type of an Alpha processor for which the macro is to generate a case table entry.

The CPUDISP macro identifies the following Alpha systems:

EV3	Reduced functionality Alpha system
EV4	Fully functional Alpha system
MANNEQUIN	Alpha simulator

continue=YES

Specifies whether execution should continue at the line immediately after the CPUDISP macro if the value at EXE$GQ_CPUTYPE does not correspond to any of the values specified as the **CPU-type** in the **list** argument. A fatal bugcheck of UNSUPRTCPU occurs if the dispatching code does not find the executing processor identified in the **list** and the value of **continue** is NO.

Description

The CPUDISP macro provides a means for transferring control to a specified destination depending on the CPU type of the executing processor.

CPUDISP constructs appropriate symbolic constants for each **CPU-type** listed in **list**, and compares them against the contents of EXE$GQ_CPUTYPE. These constants have the form HWRPB$_CPU_TYPE$K_*CPU-type*.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- With the presence of the new SYSDISP macro, the operation of the CPUDISP macro becomes less complex. The OpenVMS Alpha version of CPUDISP provides a means for transferring control to a routine entry point based solely on the type of processor chip employed in the Alpha system. The ability to dispatch specifically on the Alpha system type (or subtype, as this parameter is called in descriptions of the OpenVMS VAX version of CPUDISP) is provided on OpenVMS Alpha systems by the SYSDISP macro.
- The default value of the **continue** argument on OpenVMS Alpha systems is **YES**. In other words, CPUDISP does not request the UNSUPRTCPU bugcheck by default, should you not specify the executing CPU-type in the **list** argument.

CRAM_ALLOC

Allocates a controller register access mailbox.

Format

CRAM_ALLOC   cram [,idb] [,ucb] [,adp]

Parameters

cram
Location to which the address of the allocated CRAM is returned.

[idb]
Address of IDB for device.

[ucb]
Address of UCB for device.

[adp]
Address of ADP for device.

Description

CRAM_ALLOC allocates a controller register access mailbox (CRAM) by calling IOC$ALLOCATE_CRAM. Code must be executing at or below IPL$_SYNCH and not be holding spin locks ranked higher than IO_MISC when invoking the CRAM_ALLOC macro. For example:

```
        CRAM_ALLOC      CRAM=PDT$L_R_XBE(R4),-
                        IDB=R3,-
                        UCB=R5,-
                        ADP=R2
```

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.

Format

CRAM_CMD   index ,offset ,adp [,cram] [,command]

Parameters

index

Command index. IOC$CRAM_CMD uses this index to generate a mailbox command that is specific to the tightly-coupled interconnect that is to be the target of a request using this CRAM. You can specify any of the following values (defined by the $CRAMDEF macro), although which of these I/O operations is supported depends on the I/O interconnect that is to be the object of the mailbox operation.

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

offset

Byte offset of the field to be written or read from the base of device interface register (CSR) space. Calculation of the RBADR and MASK fields of the hardware mailbox depends on the addressing and masking mechanisms provided by the remote bus. The **byte_offset** parameter is used by IOC$CRAM_CMD to calculate the RBADR, and for write operations, is used to calculate the MASK as well.

adp

Address of ADP associated with this command. IOC$CRAM_CMD uses this parameter to determine which tightly-coupled I/O interconnect is the object of the mailbox transaction and to construct the mailbox command accordingly.

[cram]
Address of CRAM. If this parameter is specified, IOC$CRAM_CMD returns the command, mask, and remote bus address values in the corresponding fields of the hardware I/O mailbox. You must specify the **cram** argument, **command** argument, or both.

[command]
Address of buffer, two quadwords in length. If this parameter is specified, IOC$CRAM_CMD returns the command, mask, and remote bus address values in the specified buffer. You must specify the **cram** argument, **command** argument, or both.

Description

CRAM_CMD calls IOC$CRAM_CMD to generate bus-specific values for the command, mask, and remote bus fields of the hardware I/O mailbox that is the target of the mailbox operation, inserting these values into the indicated mailbox, buffer, or both.

```
CRAM_CMD        INDEX=#CRAMCMD$K_RDLONG32,-
                OFFSET=#XMI$L_XDEV,-
                ADP=R2,-
                CRAM=PDT$L_R_XDEV(R4)
```

CRAM_DEALLOC

Deallocates a controller register access mailbox.

Format

CRAM_DEALLOC cram

Parameters

cram
Address of CRAM to be deallocated by IOC$DEALLOCATE_CRAM.

Description

CRAM_DEALLOC deallocates a controller register access mailbox. When invoking the CRAM_DEALLOC macro, a device driver must be executing at or below IPL 8 and not be holding spin locks ranked higher than IO_MISC.

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.

Format

CRAM_IO   cram

Parameters

cram
Address of CRAM associated with the hardware I/O mailbox transaction.

Description

The CRAM_IO macro calls IOC$CRAM_IO to perform an entire hardware I/O mailbox transaction from the queuing of the hardware I/O mailbox to the MBPR to the transaction's completion. Invoking the CRAM_IO macro is the equivalent to successive invocations of the CRAM_QUEUE and CRAM_WAIT macros.

Prior to invoking CRAM_IO, a driver typically invokes CRAM_CMD to insert a command, mask, and remote interconnect address into the hardware I/O mailbox portion of the CRAM. For CRAMs involved in writes to device interface registers, the driver must also insert the data to be written into CRAM$Q_WDATA.

CRAM_QUEUE

Queues the hardware I/O mailbox defined within a controller register access mailbox (CRAM) to the mailbox pointer register (MBPR).

Format

CRAM_QUEUE   cram

Parameters

cram
Address of CRAM to be queued.

Description

The CRAM_QUEUE macro calls IOC$CRAM_QUEUE to initiate an I/O operation to a device in remote I/O space by writing the physical address of the hardware I/O mailbox portion of a CRAM to the MBPR. Prior to invoking CRAM_QUEUE, a driver typically invokes CRAM_CMD to insert a command, mask, and remote interconnect address into the hardware I/O mailbox portion of the CRAM. For CRAMs involved in writes to device interface registers, the driver must also insert the data to be written into CRAM$Q_WDATA,

It is expected that the driver will eventually invoke CRAM_WAIT to await completion of the request.

CRAM_WAIT

Awaits the completion of a hardware I/O mailbox transaction to a tightly-coupled I/O interconnect.

Format

CRAM_WAIT   cram

Parameters

cram
Address of CRAM associated with a previously queued hardware I/O mailbox transaction.

Description

The CRAM_WAIT macro calls IOC$CRAM_WAIT to check the done bit in the hardware I/O mailbox (CRAM$V_MBX_DONE in CRAM$W_MBX_FLAGS) and return status. It is expected that the caller has previously called IOC$CRAM_QUEUE to post to the MBPR the hardware I/O mailbox defined within the specified CRAM for an I/O operation.

DDTAB

Generates a driver dispatch table (DDT) labeled *devnam*$DDT.

Format

```
DDTAB   devnam ,[start=IOC$RETURN_SUCCESS]
        ,[ctrlinit=IOC$RETURN_SUCCESS] ,functb
        [,cancel=IOC$RETURN_SUCCESS] [,regdmp=IOC$RETURN_SUCCESS]
        [,diagbf=0] [,erlgbf=0] [,unitinit=IOC$RETURN_SUCCESS]
        [,altstart=IOC$RETURN_SUCCESS] [,mntver=IOC_STD$MNTVER]
        [,cloneducb=IOC$RETURN_SUCCESS]
        [,mntv_sssc=IOC$RETURN_SUCCESS]
        [,mntv_for=IOC$RETURN_SUCCESS]
        [,mntv_sqd=IOC$RETURN_SUCCESS]
        [,channel_assign=IOC$RETURN_SUCCESS]
        [,cancel_selective=IOC$RETURN_SUCCESS] [,kp_stack_size=0]
        [,kp_reg_mask=0] [,kp_startio=IOC$RETURN_SUCCESS] [,aux_storage=0]
        [,aux_routine=IOC$RETURN_SUCCESS] [,step]
```

Parameters

devnam
Generic name of the device.

[start=IOC$RETURN_SUCCESS]
Address of the driver's start-I/O routine. For drivers that use the kernel process services, this is the address of the kernel process start-I/O routine (EXE_STD$KP_STARTIO).

[ctrlinit=IOC$RETURN_SUCCESS]
Address of the controller initialization routine.

functb
Address of the driver's function decision table (FDT).

[cancel=IOC$RETURN_SUCCESS]
Address of the cancel-I/O routine. Many drivers specify the address of the system cancel-I/O routine (IOC_STD$CANCELIO) in this argument.

[regdmp=IOC$RETURN_SUCCESS]
Address of the routine that dumps the device registers to an error message buffer or to a diagnostic buffer.

[diagbf=0]
Length in bytes of the diagnostic buffer.

[erlgbf=0]
Length in bytes of the error message buffer.

[unitinit=IOC$RETURN_SUCCESS]
Address of the unit initialization routine.

[altstart=IOC$RETURN_SUCCESS]
Address of the alternate start-I/O routine.

[mntver=IOC_STD$MNTVER]
Address of the system-provided routine that is called at the beginning and end of a mount verification operation. The default, IOC_STD$MNTVER, is suitable for all single-stream disk drives. This argument is reserved to Digital.

[cloneducb=IOC$RETURN_SUCCESS]
Address of the routine called when a UCB is cloned by the $ASSIGN system service.

[mntv_sssc=IOC$RETURN_SUCCESS]
Address of the routine called when the system performs mount verification for a shadow set state change. This argument is reserved to Digital.

[mntv_for=IOC$RETURN_SUCCESS]
Address of the routine called when the system performs mount verification for a foreign device. This argument is reserved to Digital.

[mntv_sqd=IOC$RETURN_SUCCESS]
Address of the routine called when the system performs mount verification for a sequential device. This argument is reserved to Digital.

[channel_assign=IOC$RETURN_SUCCESS]
Address of the routine, called by SYS$ASSIGN, to complete channel assignment in a device-specific manner. This argument is reserved to Digital. (Channel-assignment routines are not yet implemented on OpenVMS Alpha systems.)

[cancel_selective=IOC$RETURN_SUCCESS]
Address of the routine that cancels a list of I/O requests from the specified channel, including both waiting and active requests. This argument is reserved to Digital. (Cancel selective routines are not yet implemented on OpenVMS Alpha systems.)

[kp_stack_size=0]
Size in bytes of the kernel process stack. EXE_STD$KP_STARTIO uses this value, or KPB$K_MIN_IO_STACK (currently 8KB), whichever is larger, to determine the size of the stack created for the driver’s start I/O kernel process thread.

[kp_reg_mask=0]
Kernel process register save mask.

This mask represents those registers used by a kernel process that must be preserved across kernel process context switches. R12 through R15, R26, R27, and R29 (KPREG$K_MIN_REG_MASK) are always preserved across kernel process context switches, and that EXE$KP_STARTIO additionally includes R2 through R5 in this register set (KPREG$K_MIN_IO_REG_MASK). R0, R1, R16 through R25, R27, R28, R30, and R31 (KPREG$K_ERR_REG_MASK) are never preserved and are illegal in a register save mask.

[kp_startio=IOC$RETURN_SUCCESS]
Address of the start-I/O routine of a driver that uses the kernel process services. Such a driver typically specifies the system routine EXE_STD$KP_STARTIO in the **start** argument to the DDTAB macro. EXE_STD$KP_STARTIO calls the start-I/O routine specified in this argument after setting up the kernel process environment.

[aux_storage=0]
Address of auxiliary storage area. This argument is reserved to Digital. (Auxiliary storage areas are not yet implemented on OpenVMS Alpha systems.)

[aux_routine=IOC$RETURN_SUCCESS]
Address of an auxiliary routine in the OpenVMS VAX mailbox driver that is called by SYS$ASSIGN. This argument is reserved to Digital. (Auxiliary routines are not yet implemented on OpenVMS Alpha systems.)

[step]
OpenVMS Alpha driver step number. You may indicate that a given driver conforms to the coding practices for an VAX OpenVMS Alpha device driver by supplying **step**=**2** in the DDTAB macro invocation. If you previously specified the **step** argument to the DPTAB macro, you need not repeat it here.

If you supply the **step** argument, but specify a value other than 1 or 2, the DPTAB macro generates the following message:

```
%MACRO-E-GENERR, Generated ERROR: DDTAB must declare driver STEP=1 or STEP=2
```

Alpha drivers typically supply a value for the **step** argument of the DPTAB macro. If the step values given the DPTAB and DDTAB macros conflict, the DDTAB macro generates the error:

```
%MACRO-E-GENERR, Generated ERROR: DDTAB STEP=x conflicts with prior declaration.
```

Description

The DDTAB macro creates a driver dispatch table (DDT), using the DRIVER_DATA macro to place it within the driver's data program section ($$$110_DATA). The macro assigns the table a label in the form of **devnam**$DDT.

The DDTAB macro writes the address of the universal executive routine vector IOC$RETURN_SUCCESS into routine address fields of the DDT that are not supplied in the macro invocation (with the exception of the **mntver** argument). IOC$RETURN_SUCCESS places success status in R0 and issues an RSB instruction.

Example

```
DDTAB      -                      ;DDT-creation macro
DEVNAM=XX, -                      ;Name of device
START=XX_START,-                  ;Start-I/O routine
FUNCTB=XX_FUNCTABLE,-             ;FDT address
CANCEL=IOC_STD$CANCELIO,-             ;Cancel-I/O routine
REGDMP=XX_REGDUMP,-               ;Register dumping routine
DIAGBF=<<15*4>+<<3+5+1>*4>>,-     ;Diagnostic buffer size
ERLGBF=<<15*4>+<1*4>+<EMB$L_DV_REGSAV>>  ;Error message buffer size
```

This code excerpt uses the DDTAB macro to create a driver dispatch table for the XX device type.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- The OpenVMS Alpha version of the DDTAB macro does not automatically define the code psect $$$115_DRIVER; rather, it invokes the DRIVER_DATA macro to include the DDT in data psect $$$110_DATA. On OpenVMS Alpha systems, you must explicitly invoke the DRIVER_CODE macro to define the $$$115_DRIVER code psect prior to the first line of executable code.
- The number and order of the arguments to the DDTAB macro are different on OpenVMS Alpha systems than on OpenVMS VAX systems.
- On OpenVMS Alpha systems, you do not distinguish (by use of the plus sign (+)) entry points of OpenVMS routines that are at absolute addresses from entry points at relative locations within the driver. For instance, an OpenVMS Alpha device driver could specify the following argument to the DDTAB macro:

  ```
  CANCEL=IOC_STD$CANCELIO,-
  ```

  It is the equivalent of the following argument specification in an OpenVMS VAX device driver:

  ```
  CANCEL=+IOC$CANCELIO,-
  ```
- An OpenVMS Alpha device driver that uses the kernel process services specifies the name of EXE_STD$KP_STARTIO in **start** argument, and the procedure value of the driver's start-I/O routine in the **kp_startio** argument.
- An OpenVMS Alpha device driver that uses the kernel process services indicates the size of the kernel mode stack in the **kp_stack_size**, and specifies a mask of registers to be preserved across kernel process context switches in the **kp_reg_mask** argument.
- Because the procedure value of the controller initialization routine is stored in the DDT (DDT$PS_CTRLINIT) in OpenVMS Alpha systems, you specify its location by using the new **ctrlinit** argument to the DDTAB macro. (On OpenVMS VAX systems, you specify the location of the controller initialization by issuing a DPT_STORE macro to VEC$L_INITIAL.)
- The OpenVMS Alpha version of the DDTAB macro does not provide the **unsolic** argument.
- Although the **channel_assign**, **cancel_selective**, **aux_storage**, and **aux_routine** arguments are allowed in the macro invocation, the functionality they represent has not yet been implemented in OpenVMS Alpha systems.
- An OpenVMS Alpha terminal port driver cannot share a single DDT with the OpenVMS Alpha terminal class driver. The terminal port driver must invoke the DDTAB macro specifying the **ctrlinit** and **unitinit** arguments. The CLASS_UNIT_INIT macro, when invoked by the port driver, initializes the remainder of the port driver's DDT from the class driver's DDT.

DEVICELOCK

Achieves synchronized access to a device's database as appropriate to the processing environment.

Format

DEVICELOCK [lockaddr] [,lockipl] [,savipl] [,condition] [,preserve=YES]

Parameters

[lockaddr]

Address of the device lock to be obtained. If **lockaddr** is not present, DEVICELOCK presumes that R5 contains the address of the UCB and uses the value at UCB$L_DLCK(R5) as the lock address.

[lockipl]

Synchronization IPL. OpenVMS Alpha always obtains this IPL from the device lock's data structure and, thus, ignores this argument.

[savipl]

Location at which to save the current IPL.

[condition]

Indication of a special use of the macro. The only defined **condition** is **NOSETIPL**, which causes the macro to omit setting IPL. In some instances, setting IPL is undesirable or unnecessary when a driver obtains a device lock. For example, when an interrupt service routine issues the DEVICELOCK macro, the dispatching of the device interrupt has already raised IPL to device IPL.

[preserve=YES]

Indication that the macro should preserve R0 across the invocation. If you do not need to retain the contents of R0, specifying **preserve**=**NO** can enhance system performance.

Description

In a *uniprocessing* environment, the DEVICELOCK macro raises IPL to the IPL indicated by the device lock's data structure (if **condition**=**NOSETIPL** is not specified).

In a *multiprocessing* environment, the DEVICELOCK macro performs the following actions:

- Preserves R0 through the macro call (if **preserve**=**YES** is specified).
- Stores the address of the device lock in R0.
- Calls either SMP$ACQUIREL or SMP$ACQNOIPL, depending upon the presence of **condition**=**NOSETIPL**. SMP$ACQUIREL raises IPL to device IPL prior to obtaining the lock, determining appropriate IPL from the device lock's data structure (SPL$B_IPL).

In both processing environments, the DEVICELOCK macro performs the following tasks:

- Preserves the current IPL at the specified location (if **savipl** is specified)
- Sets the SMP-modified bit in the driver prologue table (DPT$V_SMPMOD in DPT$L_FLAGS)

Example

```
     DEVICELOCK -
        LOCKADDR=UCB$L_DLCK(R5),- ;Lock device access
        SAVIPL=-(SP),-            ;Save current IPL
        PRESERVE=YES              ;Save R0
     SETIPL  #31                  ;Disable all interrupts
     BBC     #UCB$V_POWER,-       ;If clear - no power failure
             UCB$L_STS(R5),L1     ;...
                                  ;Service power failure!
   .
   .
   .
     DEVICEUNLOCK -
             LOCKADDR=UCB$L_DLCK(R5),- ;Unlock device access
             NEWIPL=(SP)+,-            ;Restore IPL
             PRESERVE=YES              ;Save R0
     BRW     RETREG                    ;Exit
L1:                                    ;Return for no power failure
   .
   .
   .
     WFIKPCH RETREG,#2                 ;Wait for interrupt
```

This start-I/O routine invokes the DEVICELOCK macro to synchronize access to the device's registers and UCB fields. Thus synchronized at device IPL, and holding the device lock in a VMS multiprocessing environment, the routine raises IPL to IPL$_POWER (IPL 31) to check for a power failure on the local processor. If a power failure has occurred, the routine releases the device lock and pops the saved IPL from the stack before servicing the failure. If a power failure has not occurred, the routine branches to set up the I/O request. Note that, in this instance, it is the wait-for-interrupt routine, invoked by the WFIKPCH macro, that issues the DEVICEUNLOCK macro and restores the saved IPL.

DPTAB

Generates a driver prologue table (DPT) in a program section called $$$105_PROLOGUE.

Format

```
DPTAB   [end] ,adapter ,[flags=0] ,ucbsize ,[unload] ,[maxunits=8]
        ,[defunits=1] ,[deliver] ,[vector] [,name] ,[smp=NO] ,[decode] ,step=0,
        [,idb_crams=0] [,ucb_crams=0] [,bt_order] [,ddt=DDT$BASE]
        [,struc_init=DRIVER$STRUC_INIT] [,struc_reinit=DRIVER$STRUC_REINIT]
        [,psect=$$$105_PROLOGUE] [,dpt=DRIVER$DPT]
```

Parameters

end

Unused in OpenVMS Alpha device drivers.

adapter

Type of adapter. You can supply any name that, when appended to the string "AT$_", results in a symbol defined by the $DCDEF macro in SYS$LIBRARY:STARLET.MLB. Of these symbols, the driver-loading procedure takes special action only when the keyword **NULL** is present. The driver-loading procedure creates no ADP for a null adapter (AT$_NULL) and clears the VEC$PS_ADP and IDB$L_ADP fields.

[flags=0]

Flags used in loading the driver. Drivers use the following flags:

DPT$M_SVP	Indicates that the driver requires a permanently allocated system page. Disk drivers use this SPTE during ECC correction and when using the system routines IOC_STD$MOVFRUSER and IOC_STD$MOVTOUSER. When this flag is set, the driver-loading procedure allocates a permanent system page-table entry (SPTE) for the device. It stores an index to the virtual address of the SPTE in UCB$L_SVPN when it creates the UCB. A driver can calculate the system virtual address of the page corresponding to this index by using the following formula: SVA = SEXT((LEFT_SHIFT(ucbl_svpn, mmggl_vpn_to_va)) OR va$m_system)
DPT$M_NOUNLOAD	Indicates that the driver cannot be reloaded. When this bit is set, the driver can be unloaded only by rebooting the system. Driver unloading and reloading are not supported on OpenVMS Alpha systems.

DPT$M_SMPMOD	Indicates that the driver has been designed to execute within an OpenVMS multiprocessing environment. Use of any of the multiprocessing synchronization macros (DEVICELOCK/DEVICEUNLOCK, FORKLOCK/FORKUNLOCK, or LOCK/UNLOCK) automatically sets this flag, as long as the code using the macro resides in the same module as the invocation of DPTAB.
DPT$M_DECW_DECODE	Indicates that the driver is a DECwindows class input (decoding) driver
DPT$M_NO_IDB_DISPATCH	Tells the driver-loading procedure not to create a list of UCB addresses at the end of the IDB (at IDB$L_UCBLST), regardless of the value of the **maxunits** argument or the maximum units specified in the /MAX_UNITS qualifier of the System Management (SYSMAN) utility command IO CONNECT.

ucbsize

Size in bytes of each UCB the driver-loading procedure creates for devices supported by the driver. This required argument allows drivers to extend the UCB to store device-dependent data describing an I/O operation.

[unload]

Address of the driver routine invoked by the driver-loading procedure before it unloads an old version of the driver to load a new version.

Note

The OpenVMS Alpha operating system does not yet permit driver reloading and does not support driver-unloading routines.

[maxunits=8]

Maximum number of units that this driver supports on a controller. If you omit the **maxunits** argument, the default is eight units. You can override the value specified in the DPT at driver-loading time by using the /MAX_UNITS qualifier to the SYSMAN command IO CONNECT. If DPT$M_NO_IDB_DISPATCH is not specified in the **flags** argument to the DPTAB macro, these values affect the size of the UCB list the driver-loading procedure generates at the end of the IDB.

[defunits=1]

Maximum number of UCBs to be created by the autoconfiguration facility (one for each device unit to be configured). The unit numbers assigned are zero to **defunits**–1.

If you do not specify the **deliver** argument, the autoconfiguration facility creates the number of units specified by **defunits**. If you specify the address of a unit delivery routine in the **deliver** argument, the autoconfiguration facility calls that routine to determine whether to create each UCB automatically.

[deliver]

Address of the driver unit delivery routine. The unit delivery routine determines which device units supported by this driver the autoconfiguration facility should configure automatically. If you omit the **deliver** argument, the autoconfiguration facility creates the number of units specified by the **defunits** argument.

[vector]
Address of a driver-specific transfer vector. A terminal port driver specifies the address of its vector table in this argument.

[name]
Name of the device driver. Because the OpenVMS Alpha driver-loading procedure automatically generates a driver name and writes it to the DPT, it effectively ignores this argument.

[smp=NO]
Indication of whether the driver is suitably synchronized to execute in an OpenVMS multiprocessing system. Use of any of the spin lock synchronization macros in a device driver causes the DPTAB macro to indicate multiprocessing synchronization. All OpenVMS Alpha drivers must specify **smp**=**YES**.

[decode]
Address of counted ASCII string that identifies a DECwindows class input (decoding) driver to serial-line switching code.

step
OpenVMS Alpha driver step number. You must indicate that a given driver conforms to the coding practices for an VAX OpenVMS Alpha device driver by supplying **step**=**2** in the DPTAB macro invocation. If you specify **step**=1, the macro generates the following message:

```
%MACRO-E-GENERR, Generated ERROR: *CAUTION* VAX drivers will be obsolete in V2.0
```

If you omit the **step** argument entirely, or specify a value other than 1 or 2, the DPTAB macro generates the message:

```
%MACRO-E-GENERR, Generated ERROR: DPTAB must declare driver STEP=1 or STEP=2
```

Alpha drivers may also optionally supply a value for the **step** argument of the DDTAB macro. If the step values given the DPTAB and DDTAB macros conflict, the DPTAB macro generates an error of the form:

```
%MACRO-E-GENERR, Generated ERROR: DPTAB STEP=x conflicts with prior declaration.
```

idb_crams
Number of CRAMS to be allocated and associated with the IDB. The driver-loading procedure allocates the number of CRAMs specified in **idb_crams** argument to the DPTAB macro and inserts them in the linked list headed by IDB$PS_CRAM. These CRAMs are therefore available to the driver's controller and unit initialization routine.

ucb_crams
Number of CRAMS to be allocated and associated with the UCB. The driver-loading procedure allocates the number of CRAMs specified in **ucb_crams** argument to the DPTAB macro and inserts them in the linked list headed by UCB$PS_CRAM. These CRAMs are therefore available to the driver's unit initialization routine.

[bt_order]
Ordering number for call to the runtime drivers for boot devices.

[ddt=DDT$BASE]
Address of DDT. The default is required for all devices not supplied by Digital. drivers.

[struc_init=DRIVER$STRUC_INIT]
Address of the driver I/O database initialization routine automatically generated by an invocation of the DPT_STORE macro with the **INIT** label. This routine initializes those data structure fields indicated by the invocations of the DPT_STORE macro that follow the DPT_STORE **INIT** and precede the DPT_STORE **REINIT**. The driver-loading procedure calls this initialization routine when it creates the structures and loads the driver, prior to calling the driver's controller and unit initialization routines.

The default value of this argument is required for all OpenVMS Alpha device drivers.

[struc_reinit=DRIVER$STRUC_REINIT]
Address of the driver I/O database reinitialization routine automatically generated by an invocation of the DPT_STORE macro with the **REINIT** label. This routine initializes those data structure fields indicated by the invocations of the DPT_STORE macro that follow the DPT_STORE **INIT** and precede the DPT_STORE **END**. The driver-loading procedure calls this reinitialization routine when the driver is first loaded into the system, and whenever the driver is reloaded, prior to calling the driver's controller and unit initialization routines.

The default value of this argument is required for all Alpha OpenVMS Alpha device drivers.

Note that driver unloading and reloading are not supported on OpenVMS Alpha systems.

[psect=$$$105_PROLOGUE]
Program section in which the DPT is created. The default value of this argument is required for all devices not supplied by Digital.

[,dpt=DRIVER$DPT]
Global symbol for DPT location. The default value of this argument is required for all non-Digital-supplied device drivers.

Description

The DPTAB macro, in conjunction with invocations of the DPT_STORE macro, creates a driver prologue table (DPT). The DPTAB macro places information in the DPT that allows the driver-loading procedure to identify the driver and the devices it supports. The DPTAB macro, in invoking the $SPLCODDEF definition macro, also defines the spin lock indexes used in the DPT_STORE, FORKLOCK, and LOCK macros.

Example

```
DPTAB   STEP=2,-                ;OpenVMS Alpha driver
        ADAPTER=CI,-            ;Adapter type
        UCBSIZE=UCB$C_PASIZE,-  ;UCB size
        NAME=PNDRIVER,-         ;Driver name
        SMP=YES,-               ;SMP capable
        FLAGS=<DPT$M_SCS!-      ;Driver requires SCS load,
                DPT$M_NOUNLOAD> ; cannot be reloaded
DPT_STORE INIT
DPT_STORE       UCB,UCB$B_FLCK,B,SPL$C_SCS ;SCS spinlock
DPT_STORE       UCB,UCB$L_DEVCHAR,L,<-  ;Device characteristics:
                DEV$M_SHR!-             ; Sharable
                DEV$M_AVL!-             ; Available
                DEV$M_ELG!-             ; Error logging device
                DEV$M_IDV!-             ; Input device
                DEV$M_ODV>              ; Output device
DPT_STORE       UCB,UCB$B_DIPL,B,PN_BR_LEVEL+16 ;Device interrupt IPL
DPT_STORE       UCB,UCB$B_DEVCLASS,B,-  ;Device class =
                DC$_BUS                 ; bus
DPT_STORE       UCB,UCB$L_ERTMAX,L,50   ;Retry count is 50 times
DPT_STORE       UCB,UCB$L_ERTCNT,L,50   ; without reboot of system
DPT_STORE REINIT
DPT_STORE       DDB,DDB$L_DDT,D,PN$DDT  ;DDT address
DPT_STORE_ISR   CRB$L_INTD,PN$MISC_INTERRUPT ; ISR address
DPT_STORE_ISR   CRB$L_INTD+<CRB$S_INTD>,PN$RSP_INTERRUPT
DPT_STORE       END
```

This excerpt from PNDRIVER.MAR contains the DPTAB macro and the series of DPT_STORE and DPT_STORE macros that create its driver prologue table.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- You must indicate that a given driver conforms to the coding practices for an OpenVMS Alpha device driver by supplying **step**=**2** in the **step** argument.
- A driver can request the driver-loading procedure to allocate CRAMs and associate them with the IDB or UCB by specifying the **idb_crams** and **ucb_crams** arguments.
- The OpenVMS Alpha driver-loading procedure does not support the reloading of VAX OpenVMS Alpha device drivers. It therefore ignores the **unload** argument to the DPTAB macro.
- Because the OpenVMS Alpha driver-loading procedure automatically generates a driver name and writes it to the DPT, it effectively ignores the **name** argument.
- OpenVMS Alpha ignores the **end** argument.
- The DPTAB macro, in conjunction with invocations of the DPT_STORE macro which specify the **INIT**, **REINIT**, and **END** labels, automatically generates driver structure initialization and reinitialization routines, storing their procedure values in the DPT. The default global symbol name fot the DPT location is now DRIVER$DPT instead of EVMS$DRIVER_DPT.

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.

Format

DPT_STORE str_type ,str_off ,oper ,exp [,pos] [,size]

Parameters

str_type
Type of data structure (CRB, DDB, IDB, ORB, or UCB) into which the driver-loading procedure is to store the specified data, or a label denoting a table marker. Table marker labels indicate the start of a list of DPT_STORE macro invocations that store information for the driver-loading procedure in the driver initialization table and driver reinitialization table sections of the DPT. If this argument is a table marker label, no other argument is allowed. The following labels are used:

INIT	Indicates the start of fields to initialize when the driver is loaded
REINIT	Indicates the start of additional fields to initialize when the driver is loaded and reinitialized when the driver is reloaded
END	Indicates the end of the two lists

str_off
Unsigned offset into the data structure in which the data is to be stored. This value cannot be more than 65,535 bytes.

oper
Type of storage operation, one of the following:

Type	Meaning
B	Write a byte value.
W	Write a word value.
L	Write a longword value.
D	Write an address relative to the beginning of the driver.
V	Write a bit field. If you specify a **V** in the **oper** argument, the driver-loading procedure uses the **exp**, **pos**, and **size** arguments in the bit insertion operation.

If an at sign (@) precedes the **oper** argument, the **exp** argument indicates the address of the data that is to be stored and not the data itself.

exp
Expression indicating the value with which the driver-loading procedure is to initialize the indicated field. If an at sign character (@) precedes the **oper** argument, the **exp** argument indicates the address of the data with which to initialize the field. For example, the following macro indicates that the contents of the location DEVICE_CHARS are to be written into the DEVCHAR field of the UCB.

```
DPT_STORE UCB,UCB$L_DEVCHAR,@L,DEVICE_CHARS
```

[pos]
Starting bit position within the specified field; used only if **oper**=**V**.

[size]
Number of bits to be written; used only if **oper**=**V**.

Description

The DPT_STORE macro provides a mechanism for a driver to initialize specific data structure fields when the driver is first loaded and when the driver is reloaded. A driver typically contains a series of DPT_STORE invocations which, together, automatically create a driver I/O database initialization routine and a driver I/O database reinitialization routine. The DPTAB macro writes the locations of these routines in the DPT. The driver-loading routine calls the initialization routine when a driver is first loaded; it calls the reinitialization routine both when the driver is first loaded and when the driver is reloaded. OpenVMS Alpha device drivers cannot be reloaded.

A driver constructs the initialization tables by following the DPTAB macro with one or more invocations of the DPT_STORE macro.

Drivers use the DPT_STORE macro with the **INIT** table marker label to begin a list of DPT_STORE invocations that supply initialization data for the following fields:

UCB$B_FLCK	Index of the fork lock under which the driver performs fork processing. Fork lock indexes are defined by the $SPLCODDEF definition macro (invoked by DPTAB) as follows:

IPL	Fork Lock Index
8	SPL$C_IOLOCK8
9	SPL$C_IOLOCK9
10	SPL$C_IOLOCK10
11	SPL$C_IOLOCK11

UCB$B_DIPL	Device interrupt priority level

Other commonly initialized fields are as follows:

UCB$L_DEVCHAR	Device characteristics
UCB$B_DEVCLASS	Device class
UCB$B_DEVTYPE	Device type
UCB$W_DEVBUFSIZ	Default buffer size
UCB$Q_DEVDEPEND	Device-dependent parameters

Drivers use the DPT_STORE macro with the **REINIT** table marker label to begin a list of DPT_STORE and DPT_STORE_ISR invocations that supply initialization and reinitialization data. The following fields are declared with the DPT_STORE_ISR macro:

CRB$L_INTD	Interrupt service routine
CRB$L_INTD2	Interrupt service routine for second interrupt vector

For an example of the use of the DPT_STORE macro, see the description of the DPTAB macro.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

- Because the OpenVMS Alpha driver-loading procedure automatically stores the address of the DDT in the DDB, an OpenVMS Alpha device driver does not invoke the DPT_STORE macro to write this address. For instance, the following line should be removed from an existing OpenVMS VAX driver that is to be moved to OpenVMS Alpha:

```
DPT_STORE  DDB,DDB$L_DDT,D,XA$DDT  ;Address of DDT
```

- Because the procedure value of the controller and unit initialization routines are stored in the DDT (DDT$PS_CTRLINIT and DDT$L_UNITINIT, respectively) in OpenVMS Alpha systems, you specify their location by using the **ctrlinit** and **unitinit** arguments to the DDTAB macro. The following uses of the DPT_STORE macro do not work on OpenVMS Alpha systems:

```
DPT_STORE  CRB,VEC$L_INITIAL,D,XA$CTRL_INIT  ;Address of controller init routine
DPT_STORE  CRB,VEC$L_UNITINIT,D,XA$UNIT_INIT ;Address of unit init routine
```

- Because the interrupt dispatcher requires the addresses of both the code entry point and the procedure descriptor of an interrupt service routine, you must use the new DPT_STORE_ISR macro (which generates both) to declare the routine. For instance, you should use:

```
DPT_STORE_ISR   CRB$L_INTD, XA_INTERRUPT
                                    ;Address of interrupt service routine
```

  instead of:

```
DPT_STORE  CRB,CRB$L_INTD+VEC$L_ISR,D,-
                XA_INTERRUPT       ;Address of interrupt service routine
```

- The DPTAB macro, in conjunction with invocations of the DPT_STORE macro which specify the **INIT**, **REINIT**, and **END** labels, automatically generates driver structure initialization and reinitialization routines, storing their procedure values in the DPT.

- Be aware that certain data structure fields (such as UCB$B_FIPL) have been made obsolete in OpenVMS Alpha. The names of other fields may have changed, typically to reflect a change in size of the datum.

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).

Format

DPT_STORE_ISR vec_off ,entry

Parameters

vec_off
Symbolic offset to interrupt transfer vector within the CRB. These offsets are of the following form:

Symbolic Offset	Description
CRB$L_INTD	First interrupt transfer vector
CRB$L_INTD2	Second interrupt transfer vector
CRB$L_INTD+<2*VEC$K_LENGTH>	Third interrupt transfer vector

entry
Procedure value of an interrupt service routine.

Description

The DPT_STORE_ISR macro provides a mechanism for a driver to initialize the VEC$PS_ISR_PD and VEC$PS_ISR_CODE fields of an interrupt transfer vector block (VEC) with the addresses of an interrupt service routine's procedure descriptor and code entry point, respectively. Like invocations of the DPT_STORE macro, you invoke the DPT_STORE_ISR macro within the context of the DPTAB macro.

Typically, you use DPT_STORE_ISR within the reinitialization section of the DPT (following DPT_STORE REINIT), so that the VEC fields are initialized at both driver loading and reloading.

Example

```
        DPT_STORE_ISR   CRB$L_INTD, XA_INTERRUPT
```

This invocation of the DPT_STORE_ISR macro locates the first interrupt transfer vector associated with the device controller, and places the address of XA_INTERRUPT's procedure descriptor in VEC$PS_ISR_PD and the address of its code entry point in VEC$PS_ISR_CODE.

$DRIVER_ALTSTART_ENTRY

Format

$DRIVER_ALTSTART_ENTRY   PRESERVE=<R2,R3,R4,R5>,FETCH=YES

```
  $OFFDEF ALTARG, < -
   irp, -
   ucb >

Parameter offsets:
  MOVL ALTARG$_IRP(AP), R3
  MOVL ALTARG$_UCB(AP), R5
```

$DRIVER_CANCEL_ENTRY

Format

$DRIVER_CANCEL_ENTRY PRESERVE=<R2,R3,R4>, FETCH=YES

```
  $OFFDEF CANARG, < -
   chan,-
   irp,-
   pcb,-
   ucb,-
   reason >

Parameter offsets:
  MOVL CANARG$_CHAN(AP), R2
  MOVL CANARG$_IRP(AP), R3
  MOVL CANARG$_PCB(AP), R4
  MOVL CANARG$_UCB(AP), R5
  MOVL CANARG$_REASON(AP), R8
```

$DRIVER_CANCEL_SELECTIVE

Format

$DRIVER_CANCEL_SELECTIVE_ENTRY   PRESERVE, FETCH=YES

```
  $OFFDEF CANSARG, < -
   pcb, -
   ucb, -
   chan, -
   iosb_vector, -
   iosb_count >
Parameter offsets:

                MOVL    #SS$_UNSUPPORTED, R0
  MOVL CANSARG$_PCB(AP), R4
  MOVL CANSARG$_UCB(AP), R5
  MOVL CANSARG$_CHAN(AP), R6
  MOVL CANSARG$_IOSB_VECTOR(AP), R7
  MOVL CANSARG$_IOSB_COUNT(AP), R8
```

$DRIVER_CHANNEL_ASSIGN

Format

$DRIVER_CHANNEL_ASSIGN_ENTRY   PRESERVE, FETCH=YES

```
  $OFFDEF CHANARG, < -
   ucb, -
   ccb >
Parameter offsets:
  MOVL CHANARG$_UCB(AP), R5
  MOVL CHANARG$_CCB(AP), R8
```

$DRIVER_CLONEDUCB

Format

$DRIVER_CLONEDUCB   PRESERVE=R3, FETCH=YES

```
  $OFFDEF CLONEARG, < -
   cloned_ucb,-
   ddt,-
   pcb,-
   template_ucb >
Parameter offsets:
                MOVL    #SS$_NORMAL, R0
  MOVL CLONEARG$_CLONED_UCB(AP), R2
  MOVL CLONEARG$_DDT(AP), R3
  MOVL CLONEARG$_PCB(AP), R4
  MOVL CLONEARG$_TEMPLATE_UCB(AP), R5
```

DRIVER_CODE

Declares the program section (psect) that contains driver code.

Format

DRIVER_CODE [pname=$$$115_DRIVER]

Parameters

[pname=$115_DRIVER]
Name of driver psect that contains driver code. The default psect name, $115_DRIVER, is suitable for most temporary OpenVMS Alpha drivers, although you can specify an alternative name.

Description

The DRIVER_CODE macro generates a psect for driver code, with attributes that allow the Linker utility (linker) to properly and compatibly collect driver image sections into a loadable executive image.

You must precede the first line of executable code in a Step 1 OpenVMS Alpha device driver with an invocation of the DRIVER_CODE macro. If the driver consists of multiple source modules, you should replace each explicit setting of the $$$115_DRIVER psect with an invocation of this macro to ensure that the correct standard psect for driver code sections is always used.

OpenVMS driver macros that construct driver code automatically invoke the DRIVER_CODE macro prior to creating the code. For instance, the DPT_STORE macro automatically invokes the DRIVER_CODE macro prior to constructing the driver initialization and reinitialization routines.

Note

Use of the DRIVER_CODE macro requires that you define the symbol "EVAX".

$DRIVER_CRTLINIT

Format

$DRIVER_CTRLINIT_ENTRY   PRESERVE=R2, FETCH=YES

```
  $OFFDEF CTRLARG, < -
   idb, -
   ddb, -
   crb >
Parameter offsets:
                MOVL    #SS$_NORMAL, R0
  MOVL CTRLARG$_IDB(AP), R4
  MOVL CTRLARG$_IDB(AP), R5
  MOVL CTRLARG$_DDB(AP), R6
  MOVL CTRLARG$_CRB(AP), R8
```

$DRIVER_DELIVER_ENTRY

Format

$DRIVER_DELIVER_ENTRY PRESERVE=<R2>, FETCH=YES

```
  $OFFDEF DLVRARG, < -
   idb, -
   unit_number, -
   scratch_area, -
   adp >
```

```
Parameter offsets:
  MOVL DLVRARG$_IDB(AP), R3
  MOVL DLVRARG$_IDB(AP), R4
  MOVL DLVRARG$_UNIT_NUMBER(AP), R5
  MOVL DLVRARG$_SCRATCH_AREA(AP), R7
  MOVL DLVRARG$_ADP(AP), R8
```

$DRIVER_ERRRTN

Format

$DRIVER_ERRRTN_ENTRY   PRESERVE, FETCH=YES

```
  $OFFDEF ERRARG, < -
   irp, -
   pcb, -
   ucb, -
   ccb, -
   status>
Parameter offsets:
  MOVL ERRARG$_IRP(AP),R3
  MOVL ERRARG$_PCB(AP),R4
  MOVL ERRARG$_UCB(AP),R5
  MOVL ERRARG$_CCB(AP),R6
  MOVL ERRARG$_STATUS(AP),R0
```

$DRIVER_FDT_ENTRY

Format

$DRIVER_FDT_ENTRY PRESERVE=<R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,
R13,R14,R15>, FETCH=YES

```
Parameter offsets:
 MOVL FDTARG$_IRP(AP),R3
 MOVL FDTARG$_PCB(AP),R4
 MOVL FDTARG$_UCB(AP),R5
 MOVL FDTARG$_CCB(AP),R6
```

$DRIVER_MNTVER

Format

$DRIVER_MNTVER_ENTRY PRESERVE, FETCH=YES

```
  $OFFDEF MNTARG, < -
   irp, -
   ucb >

Parameter offsets:
  MOVL MNTARG$_IRP(AP), R3
  MOVL MNTARG$_UCB(AP), R5
```

$DRIVER_REGDUMP

Format

$DRIVER_REGDUMP_ENTRY   PRESERVE=<R2>, FETCH=YES

```
  $OFFDEF REGARG, < -
   buffer, -
   cram, -
   ucb >
```

```
Parameter offsets:
  MOVL REGARG$_BUFFER(AP), R0
  MOVL REGARG$_CRAM(AP), R4
  MOVL REGARG$_UCB(AP), R5
```

$DRIVER_START_ENTRY

Format

$DRIVER_START_ENTRY   PRESERVE=<R2,R4>, FETCH=YES

```
  $OFFDEF STARTARG, < -
   irp, -
   ucb >
```

```
Parameter offsets:
  MOVL STARTARG$_IRP(AP), R3
  MOVL STARTARG$_UCb(AP), R5
```

$DRIVER_UNITINIT

Format

$DRIVER_UNITINIT_ENTRY   PRESERVE=<R2>, FETCH=YES

```
  $OFFDEF UNITARG, < -
   idb, -
   ucb >
Parameter offsets:
                MOVL    #SS$_NORMAL, R0
  MOVL UNITARG$_IDB(AP), R4
  MOVL UNITARG$_UCB(AP), R5
```

DRIVER_DATA

Declares the program section (psect) that contains driver data.

Format

DRIVER_DATA [pname=$$$110_DATA]

Parameters

[pname=$110_DATA]
Name of driver psect that contains driver data. The default psect name, $110_DATA, is suitable for most temporary OpenVMS Alpha drivers, although you can specify an alternative name.

Description

The DRIVER_DATA macro generates a psect for driver data, with attributes that allow the Linker to properly and compatibly collect driver image sections into a loadable executive image. You must precede any driver data by an invocation of this macro.

OpenVMS driver macros that construct data, such as DDTAB and FUNCTAB, automatically invoke the DRIVER_DATA macro prior to creating the data.

$FDTARGDEF

Format

$FDTARGDEF

```
$OFFDEF FDTARG,<IRP,PCB,UCB,CCB>
```

FDT_ACT

Initializes the FDT action routine vector slot corresponding to one or more specified I/O function codes with the procedure value of the specified upper-level FDT action routine.

Format

FDT_ACT   action, codes

Parameters

action
Action routine that services the I/O function codes identified by the **codes** argument.

codes
List of codes (enclosed within angle brackets and separated by commas) for I/O functions serviced by the specified upper-level FDT action routine. The macro expansion prefixes each code with the string IO$_; for example, READVBLK expands to IO$_READVBLK.

Description

The FDT_ACT macro identifies the upper-level FDT action routine that processes one or more specified I/O function codes. If, at the time it invokes FDT_ACT, the driver has not yet invoked the FDT_INI macro, FDT_ACT invokes it on the driver's behalf, creating an FDT with the label DRIVER$FDT.

An OpenVMS Alpha device driver specifies one or more legal I/O functions by supplying the address of an upper-level FDT action routine for that function to the FDT_ACT macro. The FDT_ACT macro initializes the slot in the FDT action routine vector corresponding to each supplied function code with the procedure value of the specified routine.

Multiple invocations of the FDT_ACT macro, in sum, define the full set of I/O functions serviced by the driver. An illegal I/O function is one that the driver does not list in any FDT_ACT macro invocations. Its vector slot contains the procedure value of the illegal I/O function processing routine (EXE$ILLIOFUNC).

Note, however, only one upper-level FDT action routine can service any given I/O function. If you reuse an I/O function code in an FDT_ACT invocation, the compiler generates an error of the form:

```
%MACRO-E-GENERR, Generated ERROR: Multiple actions defined for function IO$_xxxxxx
```

A consequence of this limitation is that, if the preprocessing of a given function requires that several routines be executed, the upper-level FDT action routine must set up the appropriate call chain.

Example

```
XX_FUNCTABLE:                           ;Function decision table
     FDT_INI   XX$FDT
     FDT_BUF   -                        ;Buffered-I/O functions
               <READLBLK,-              ;Read logical block
                READPBLK,-              ;Read physical block
                READVBLK,-              ;Read virtual block
                SENSEMODE,-             ;Sense reader mode
                SENSECHAR,-             ;Sense reader characteristics
                SETMODE,-               ;Set reader mode
                SETCHAR,-               ;Set reader characteristics
               >
     FDT_ACT   XX_READ,-                ;Read function FDT routine
               <READLBLK,-              ;Read logical block
                READPBLK,-              ;Read physical block
                READVBLK,-              ;Read virtual block
               >
     FDT_ACT   EXE_STD$SETMODE,-        ;Set mode/characteristics FDT routine
               <SETCHAR,-               ;Set reader characteristics
                SETMODE,-               ;Set reader mode
               >
     FDT_ACT   EXE_STD$SENSEMODE,-      ;Sense mode/characteristics FDT routine
               <SENSECHAR,-             ;Sense reader characteristics
                SENSEMODE,-             ;Sense reader mode
               >
```

This function decision table (FDT) specifies that the routine XX_READ be called for all read functions that are valid for the device. XX_READ appears later in the driver module. System I/O preprocessing will call routines EXE_STD$SETMODE and EXE_STD$SENSEMODE for the device's set-characteristics and sense-mode functions.

FDT_BUF

Builds the buffered function mask within a driver's function decision table (FDT) from the specified list of I/O functions.

Format

FDT_BUF [codes]

Parameters

[codes]
List of codes (enclosed within angle brackets and separated by commas) for I/O functions supported by the driver that require an intermediate system buffer. The macro expansion prefixes each code with the string IO$_; for example, READVBLK expands to IO$_READVBLK.

Description

The FDT_BUF macro builds the buffered function mask within an FDT from the specified list of I/O functions.

An OpenVMS Alpha device driver invokes the FDT_BUF macro to indicate which of the I/O functions it supports require a system buffer. If the driver has not yet invoked the FDT_INI macro, FDT_BUF invokes it on the driver's behalf, creating an FDT with the label DRIVER$FDT.

A driver specifies a legal I/O function by supplying the address of an upper-level FDT action routine for that function to the FDT_ACT macro. Beware of specifying a function code in an FDT_BUF invocation that you do not also specify in an FDT_ACT invocation. The FDT action routine vector slot for such a function contains a pointer to the illegal I/O function processing routine (EXE$ILLIOFUNC).

An example of the use of FDT_BUF appears in the description of the FDT_ACT macro.

FDT_INI

Creates, labels, and initializes a function decision table (FDT).

Format

FDT_INI [fdt=DRIVER$FDT]

Parameters

[fdt=DRIVER$FDT]
Label of the start of the FDT.

Description

The FDT_INI macro creates an FDT, using the DRIVER_DATA macro to place it within the driver's data program section ($$$110_DATA). The macro properly aligns the FDT in memory, assigning it the label specified by the **fdt** argument.

FDT_INI initializes the FDT by clearing the buffered function mask and entering the address of the illegal I/O function processing routine (EXE$ILLIOFUNC) in all FDT action routine vector slots.

An OpenVMS Alpha device driver invokes the FDT_BUF macro to indicate which of the I/O functions it supports require a system buffer. A driver specifies a legal I/O function by supplying the address of an upper-level FDT action routine for that function to the FDT_ACT macro.

An example of the use of FDT_INI appears in the description of the FDT_ACT macro.

FORK

Creates a simple fork process on the local processor.

Format

FORK [routine] [,continue] [environment=JSB|CALL]

Parameters

[routine]
Name of the routine to be executed in fork context. If you omit this argument, the FORK macro assumes that the fork routine immediately follows the invocation.

[,continue]
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 macro.

[,environment
Keyword that specifies the fork routine environment as either JSB or CALL. The default is JSB. If specified as JSB, then EXE$PRIMITIVE_FORK is called and a .JSB_ENTRY directive is used to generate the fork routine. If specified as CALL, then EXE_STD$PRIMITIVE_FORK is called, a .CALL_ENTRY directive is used to generate the fork routine, the FR3, FR4, and FKB parameters in the fork routine are copied into R3, R4, and R5.

Description

The FORK macro creates a fork process. When the FORK macro is invoked, the following registers must contain the values listed:

Register	Contents
R3	Contains the 64-bit value to pass to the fork routine via FKB$Q_FR3(R5)
R4	Contains the 64-bit value to pass to the fork routine via FKB$Q_FR4(R5)
R5	Contains a pointer to the fork block

Unlike the IOFORK macro, the FORK macro does not disable device timeouts by clearing the UCB$V_TIM bit in the field UCB$L_STS.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

```
Implicit outputs to caller:
  ENVIRONMENT=CALL
    R0,R1        are scratched.
  ENVIRONMENT=JSB
    R3,R4        outputs from EXE$PRIMITIVE_FORK.
    R0,R1        are preserved.
Implicit outputs to fork routine, i.e. entry conditions:
  ROUTINE=routine_name
                 If the routine name is specified  then  the  fork
                 entry  point  is  assumed  to  be  at  the  named
                 location and no fork entry point is defined here.
                 The  named  fork  routine  can use either the new
                 standard call interface or  the  traditional  JSB
                 interface  as described in section 4.2 regardless
                 of the setting of the ENVIRONMENT keyword.
  ROUTINE=<not specified>,ENVIRONMENT=CALL
                 A fork routine entry point  is  generated  for  a
                 routine  using the new standard call interface as
                 described in section 4.2.
    R3,R4,R5     contain traditional fork routine parameter values
                 copied  from  the  standard call interface actual
                 parameters,
    R0,R1        can be scratched.
  ROUTINE=<not specified>,ENVIRONMENT=JSB
                 A fork routine entry point  is  generated  for  a
                 routine  using  the  traditional JSB interface as
                 described in section 4.2.
    R3,R4,R5     contain traditional fork routine parameters,
    R0-R4        can be scratched.
```

FORK_ROUTINE

Defines the entry point of a fork routine.

Format

FORK_ROUTINE [name=fork_routine_name] [,symbol=LOCAL | GLOBAL]
[,environment=JSB | CALL] [,fetch=YES | NO]

Parameters

[name]
Name of the fork routine.

[,symbol]
Specifies if the routine name should be declared as a local or global symbol. The default is for a local symbol.

[,environment]
Specifies the fork routine environment as either JSB or CALL. If specified as JSB, then a .JSB_ENTRY directive is used to define the fork routine entry point. If specified as CALL, then a .CALL_ENTRY directive is used to define the fork routine entry point. The default is JSB.

[,fetch]
Specifies if the fork routine parameters for an ENVIRONMENT=CALL fork routine should be copied into the traditional R3. R4. and R5 register The default is YES.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

```
Implicit inputs:

  None.

Implicit outputs, i.e.  fork routine entry conditions:

  ENVIRONMENT=CALL

    FORKARG$_FR3(AP), FORKARG$_FR4(AP), FORKARG$_FKB(AP)
                 the symbolic parameter offsets  are  defined  and
                 can   be   used   to   access  the  fork  routine
                 parameters,

    R3,R4,R5     contain traditional fork  routine  parameters  if
                 FETCH=YES,

    R0,R1        can be scratched.

  ENVIRONMENT=JSB

    R3,R4,R5     contain traditional fork routine parameters,

    R0-R4        can be scratched.
```

FORK_WAIT

Inserts a fork block on the fork-and-wait queue.

Format

FORK_WAIT [routine] [,continue] [,environment=JSB | CALL]

Parameters

[routine]
Name of the routine to be executed in fork context. If you omit this argument, the FORK_WAIT macro assumes that the fork routine immediately follows the invocation.

[,continue]
Label where execution continues after the fork block has been inserted on the fork-and-wait queue. If you omit this argument, control returns to the caller of the routine that invoked the FORK_WAIT macro.

[,environment]
Specifies the fork routine environment as either JSB or CALL. The default is JSB. If specified as JSB, then EXE$PRIMITIVE_FORK_WAIT is called and a .JSB_ENTRY directive is used to generate the fork routine. If specified as CALL, then EXE_STD$PRIMITIVE_FORK_WAIT is called, a .CALL_ENTRY directive is used to generate the fork routine, the FR3, FR4, and FKB parameters in the fork routine are copied into R3, R4, and R5.

Description

The FORK_WAIT macro inserts a fork block on the system fork-and-wait queue. When the FORK_WAIT macro is invoked, the following registers must contain the values listed:

Register	Contents
R3	Contains the 64-bit value to pass to the fork routine via FKB$Q_FR3(R5)
R4	Contains the 64-bit value to pass to the fork routine via FKB$Q_FR4(R5)
R5	Contains a pointer to the fork block

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

```
Implicit outputs to caller:
  ENVIRONMENT=CALL
    R0,R1        are scratched.
  ENVIRONMENT=JSB
    R0,R1        are preserved.
Implicit outputs to fork routine, i.e. entry conditions:
  ROUTINE=routine_name
```

If the routine name is specified then the fork entry point is assumed to be at the named location and no fork entry point is defined here. The named fork routine can use either the new standard call interface or the traditional JSB interface as described in section 4.2 regardless of the setting of the ENVIRONMENT keyword.

ROUTINE=<not specified>,ENVIRONMENT=CALL

A fork routine entry point is generated for a routine using the new standard call interface as described in section 4.2.

R3,R4,R5	contain traditional fork routine parameter values copied from the standard call interface actual parameters,
R0,R1	can be scratched.

ROUTINE=<not specified>,ENVIRONMENT=JSB

A fork routine entry point is generated for a routine using the traditional JSB interface as described in section 4.2.

R3,R4,R5	contain traditional fork routine parameters,
R0-R4	can be scratched.

FORKLOCK

Achieves synchronized access to a device driver's fork database as appropriate to the processing environment.

Format

FORKLOCK [lock] [,lockipl] [,savipl] [,preserve=YES]

Parameters

[lock]

Index of the fork lock to be obtained. If the **lock** argument is not present in the macro invocation, FORKLOCK presumes that R5 contains the address of the fork block and uses the value at FKB$B_FLCK(R5) as the lock index.

[lockipl]

Synchronization IPL. OpenVMS Alpha obtains this IPL from the spin lock data structure or spin lock IPL vector and ignores this argument.

[savipl]

Location at which to save the current IPL.

[preserve=YES]

Indication that the macro should preserve R0 across the invocation. If you do not need to retain the contents of R0, specifying **preserve**=**NO** can enhance system performance.

Description

In a *uniprocessing* environment, the FORKLOCK macro raises IPL to the IPL indicated by the entry in the spin lock IPL vector (SMP$AL_IPLVEC) that corresponds to the fork lock index.

In a *multiprocessing* environment, the FORKLOCK macro stores the fork lock index in R0 and calls SMP$ACQUIRE. SMP$ACQUIRE uses the value in R0 to locate the fork lock structure in the system spin lock database (a pointer to which is located at SMP$AR_SPNLKVEC). Prior to securing the fork lock, SMP$ACQUIRE raises IPL to its associated IPL (SPL$B_IPL).

In both processing environments, the FORKLOCK macro performs the following tasks:

- Preserves R0 through the macro call (if **preserve**=**YES** is specified)
- Preserves the current IPL at the specified location (if **savipl** is specified)
- Sets the SMP-modified bit in the driver prologue table (DPT$V_SMPMOD in DPT$L_FLAGS)

Notes for Converting VAX Drivers

If you are converting an OpenVMS VAX driver to an Alpha driver, note the following:

- Because OpenVMS Alpha obtains this IPL from the spin lock IPL vector or the spin lock data structure that corresponds to the fork lock index, it ignores the **lockipl** argument, if specified.
- Because OpenVMS Alpha drivers must use multiprocessing synchronization semantics, the **fipl** argument to the FORKLOCK macro has been removed.

IOFORK

Creates a fork process on the local processor for a device driver, disabling timeouts from the associated device.

Format

IOFORK [routine] [,continue] [,ENVIRONMENT=JSB|CALL]

Parameters

[routine]
Name of the routine to be executed in fork context. If you omit this argument, the IOFORK macro assumes that the fork routine immediately follows the invocation.

[,continue]
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 IOFORK macro.

[,environment
Keyword that specifies the fork routine environment as either JSB or CALL. The default is JSB. If specified as JSB, then EXE$PRIMITIVE_FORK is called and a .JSB_ENTRY directive is used to generate the fork routine. If specified as CALL, then EXE_STD$PRIMITIVE_FORK is called, a .CALL_ENTRY directive is used to generate the fork routine, the FR3, FR4, and FKB parameters in the fork routine are copied into R3, R4, and R5.

Description

The IOFORK macro disables device timeouts by clearing the UCB$V_TIM bit in the field UCB$L_STS and creates a fork process. When the IOFORK macro is invoked, the following registers must contain the values listed:

Register	Contents
R3	Contents to be placed in R3 of the fork process (64 bits)
R4	Contents to be placed in R4 of the fork process (64 bits)
R5	Address of fork block

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

```
Implicit outputs to caller:
  ENVIRONMENT=CALL
    R0,R1       are scratched.
  ENVIRONMENT=JSB
    R3,R4       outputs from EXE$PRIMITIVE_FORK.
    R0,R1       are preserved.
Implicit outputs to fork routine, i.e. entry conditions:
  ROUTINE=routine_name
```

If the routine name is specified then the fork entry point is assumed to be at the named location and no fork entry point is defined here. The named fork routine can use either the new standard call interface or the traditional JSB interface as described in section 4.2 regardless of the setting of the ENVIRONMENT keyword.

ROUTINE=<not specified>,ENVIRONMENT=CALL

A fork routine entry point is generated for a routine using the new standard call interface as described in section 4.2.

R3,R4,R5	contain traditional fork routine parameter values copied from the standard call interface actual parameters,
R0,R1	can be scratched.

ROUTINE=<not specified>,ENVIRONMENT=JSB

A fork routine entry point is generated for a routine using the traditional JSB interface as described in section 4.2.

R3,R4,R5	contain traditional fork routine parameters,
R0-R4	can be scratched.

IFNORD, IFNOWRT, IFRD, IFWRT

Determine the read or write accessibility of a range of memory locations.

Format

$$\left\{\begin{array}{l}\text{IFNORD}\\ \text{IFNOWRT}\\ \text{IFRD}\\ \text{IFWRT}\end{array}\right\}\quad \text{siz ,adr ,dest ,[mode=\#0] [,prvmod] [,page] [,page\_store]}$$

Parameters

siz
Offset of the last byte to check from the first byte to check, a number less than or equal to 512.

adr
Address of first byte to check.

dest
Address to which the macro transfers control, according to the following conditions:

Macro	Condition
IFNORD	If either of the specified bytes cannot be read in the specified access mode
IFNOWRT	If either of the specified bytes cannot be written in the specified access mode
IFRD	If both bytes can be read in the specified access mode
IFWRT	If both bytes can be written in the specified access mode

[mode=#0]
Mode in which access is to be checked; zero, the default, causes the check to be performed in the mode contained in the previous-mode field of the current PSL.

[prvmod]
Known previous mode of the processor, extracted from the processor status (PS).

[page]
Shifted base address of a known accessible page. The value you specify for the **page** argument can either be zero, or the value returned in the buffer specified as the **page_store** argument in a previous invocation of the macro.

[page_store]
Address of location in which the macro returns the shifted base address of the last page probed.

Description

The IFNORD and IFRD macros use the PROBER instruction to check the read accessibility of the specified range of memory by checking the accessibility of the first and last bytes in that range. The IFNORD macro passes control to the specified destination if either of the specified bytes cannot be read in the specified access mode. The IFRD macro transfers control if both bytes can be read in the specified access mode. Otherwise, the macros transfer to the next inline instruction.

The IFNOWRT and IFWRT macros use the PROBEW instruction to check the write accessibility of the specified range of memory by checking the accessibility of the first and last bytes in that range. The IFNOWRT macro passes control to the specified destination if either of the specified bytes cannot be written in the specified access mode. The IFWRT macro transfers control to the specified destination if both bytes can be written in the specified access mode. Otherwise, the macros transfer to the next in-line instruction.

On OpenVMS Alpha systems each VAX PROBE instruction generates two PALcode calls—one to read the processor status (PS) to obtain the previous processor mode and one to perform the actual probe. In modules that perform many probes—for instance, code that verifies the accessibility of an item list—these macros provide the following opimizations:

- Because the previous PS does not change in single-threaded kernel mode code, such code can store the previous mode value and reuse it for each probe operation. The **prvmod** argument is available for this purpose.
- Because all of the user's buffers are within the same CPU-specific page, particularly when processing item lists, modules that store the base address of a known accessible page, can compare buffer addresses against this base and avoid any PALcode calls. The **page** and **page_store** arguments are available for this purpose.

When processing an item list, specify the same storage location for both the **page** and **page_store** arguments in each probe macro invocation. This keeps the known accessible page base updated. If the item list does cross a page boundary, the probe operation will be performed only the one item that actually crosses the boundary; subsequent items will share the updated page base value and do not require probing.

When probing a buffer specified by an item descriptor, use the **page** argument with the known probed page, but do not use the **page_store** argument. Other items in the item list are likely to reside within the last known probed page. If the buffer is not, omitting the **page_store** argument allows you to avoid overwriting the last known probed page and issuing an Alpha PALcode call when you process subsequent items in the item list.

If you specify zero in the **page** argument as the page base address, these macros skip page base comparison. This is useful in routines that probe a number of input parameters which may or may not be present.

If a routine is probing a number of input parameters which may or may not be present, it should specify a zero in the **page** argument and clear the location pointed to by the **page_store** argument. When the **page** argument is zero, the macros skip page base comparison. In the event an argument is missing, the cleared **page_store** location allows subsequent probe macro invocationss to forego checking that location before using its value in the **page** argument.

CAUTION

These macros expect you to keep known readable pages separate from known writable pages.

Example

```
     MOVZWL    $SS_ACCVIO,R0            ;Assume read access failure
     MOVL      ENTRY_LIST(AP),R11       ;Get address of entry point list
     IFRD      #4*4,(R11),50$           ;Branch forward if process
                                        ; has read access
     BRW      ERROR                     ;Otherwise stop with error
   .
   .
   .
```

The connect-to-interrupt driver uses the IFRD macro to verify that the process has read access to the four longwords that make up the entry point list. The address of the entry point list was specified in the **p2** argument of the $QIO request to the driver.

Notes for Converting VAX Drivers

If you are converting an OpenVMS VAX driver to an Alpha driver, note that the OpenVMS Alpha versions of these macros provide optional arguments (**prvmod**, **page**, and **page_store**) that allow you to optimize code that performs many probes.

KP_ALLOCATE_KPB

Creates a KPB and a kernel process stack, as required by the Open VMS kernel process services.

Format

KP_ALLOCATE_KPB   kpb [,stack=#1024] [,flags] [,param]

Parameters

kpb
Address of KPB.

[stack=#1024]
Requested size (in bytes) of kernel process stack.

[flags]
Flags indicating the type, size, and configuration of the KPB to be created. KP_ALLOCATE_KPB accepts only the following flags:

KPB$V_VEST	KPB is a VEST KPB. (See Chapter 10 for a description of VEST KPBs.)
KPB$V_SPLOCK	Spin lock area is present. (EXE$KP_ALLOCATE_KPB automatically sets this bit when KPB$V_VEST is set.)
KPB$V_DEBUG	Debug area is present.
KPB$V_DEALLOC_AT_END	KP_END should call KP_DEALLOCATE.

[param]
Size in bytes of KPB parameter area, if any.

Description

The KP_ALLOCATE_KPB macro calls EXE$KP_ALLOCATE_KPB to create the KPB and the kernel process stack needed by a kernel process. When a driver invokes KP_ALLOCATE_KPB it cannot be executing above IPL$_SYNCH or be holding any spin locks that have higher rank than the MMG spin lock.

KP_DEALLOCATE_KPB

Deallocates a KPB and its associated kernel process stack.

Format

KP_DEALLOCATE_KPB   kpb

Parameters

kpb
Address of KPB.

Description

The KP_DEALLOCATE_KPB macro calls EXE$KP_DEALLOCATE_KPB to deallocate the KPB and the associated kernel process stack. When a driver invokes KP_DEALLOCATE_KPB, it cannot be executing above IPL$_SYNCH or be holding any spin locks of higher rank than MMG.

KP_END

Terminates the execution of a kernel process.

Format

KP_END kpb

Parameters

kpb
Address of KPB.

Description

The KP_END macro calls EXE$KP_END to terminate the execution of a kernel process and, if KPB$V_DEALLOC_AT_END in KPB$IS_FLAGS is set, to deallocate its KPB. When a driver invokes the KP_END macro, it must be executing at IPL$_RESCHED or above.

KP_RESTART

Resumes the execution of a kernel process.

Format

KP_RESTART kpb

Parameters

kpb
Address of KPB.

Description

The KP_RESTART macro calls EXE$KP_RESTART to restart a kernel process. The caller of EXE$KP_RESTART, usually a kernel process scheduling stall routine, must be executing at IPL$_RESCHED or above.

KP_REQCOM

Invokes OpenVMS device-independent I/O postprocessing from a kernel process.

Format

KP_REQCOM

Description

The KP_REQCOM macro issues a JSB instruction to IOC$REQCOM to complete the processing of an I/O request after a kernel process within a driver has finished its portion of I/O postprocessing. (The REQCOM macro cannot be used within the context of a kernel process.)

When the KP_REQCOM macro is invoked, the following registers must contain the following values:

Register	Contents
R0	First longword of I/O status
R1	Second longword of I/O status
R5	Address of UCB

The KP_REQCOM macro destroys the contents of R0 through R3. All other registers are also destroyed if the action of the macro initiates the processing of a waiting I/O request for the device.

KP_STALL_FORK, KP_STALL_IOFORK

Stall a kernel process in such a manner that it can be resumed by the OpenVMS fork dispatcher.

Format

KP_STALL_FORK   kpb [,fkb=KPB$PS_FQFL]

KP_STALL_IOFORK   [kpb=IRP$PS_KPB] [,fkb=UCB$L_FQFL]

Parameters

kpb
Address of KPB (which must be a VEST KPB). KPB$PS_UCB must contain the address of a UCB and KPB$PS_IRP must contain the address of an IRP.

KP_STALL_FORK requires a value for this argument.

[fkb]
Address of a fork block.

Description

The KP_STALL_FORK and KP_STALL_IOFORK macros stall a kernel process by calling EXE$KP_FORK.

Prior to calling IOC$KP_FORK, the KP_STALL_IOFORK macro disable timeouts from the device represented by the UCB associated with the kernel process by clearing UCB$V_TIM in UCB$L_STS.

The macros can only be called by a kernel process.

KP_STALL_FORK_WAIT

Stalls a kernel process so that it can be resumed by the software timer interrupt service routine's examination of the fork-and-wait queue.

Format

KP_STALL_FORK_WAIT   kpb [,fkb]

Parameters

kpb
Address of the caller's KPB.

[fkb]
Address of a fork block. If this argument is omitted, EXE$KP_FORK_WAIT uses the fork block within the KPB (KPB$PS_FKBLK).

Description

The KP_STALL_FORK_WAIT macro stalls a kernel process by calling EXE$KP_FORK_WAIT. Only a kernel process executing at or above IPL$_SYNCH can invoke KP_STALL_FORK_WAIT.

KP_STALL_GENERAL

Stalls the execution of a kernel process.

Format

KP_STALL_GENERAL kpb ,stall_routine [,resume_routine]

Parameters

kpb

Register containing address of the caller's KPB.

stall_routine

Procedure value of the routine to be called requested to suspend the kernel process described by the specified **kpb**.

A kernel process scheduling stall routine preserves kernel process context not represented on the kernel process stack and takes steps that allow the stalled kernel process thread to be resumed at some later time (for instance, by inserting a fork block on a fork queue or by making a timer queue entry).

At the time a kernel process scheduling stall routine is called, kernel process context has been stored in the KPB and on the kernel process stack. The stall routine can thus immediately resume the kernel process thread.

[resume_routine]

Procedure value of the routine to be invoked by EXE$KP_RESTART when a stalled kernel process is to be resumed.

Description

The KP_STALL_GENERAL macro calls EXE$KP_STALL_GENERAL to suspend execution of the current kernel process. A kernel process invokes KP_STALL_GENERAL directly — instead of KP_STALL_FORK, KP_STALL_FORK_WAIT, KP_STALL_IOFORK, KP_STALL_REQCHAN, KP_STALL_WFIKPCH, or KP_STALL_WFIRLCH — when it requires a specialized scheduling stall routine or scheduling restart routine.

Only a kernel process can invoke the KP_STALL_GENERAL macro.

KP_STALL_REQCHAN

Stalls a kernel process in such a manner that it can be resumed by the granting of a device controller channel.

Format

KP_STALL_REQCHAN [pri=LOW] [,kpb=IRP$PS_KPB] [,idb=YES]

Parameters

[pri=LOW]
Priority of the request for the controller channel. You can specify one of the following keywords:

Keyword	Meaning
LOW	Insert fork block of UCB requesting controller channel at the tail of the channel-wait queue.
HIGH	Insert fork block of UCB requesting controller channel at the head of the channel-wait queue.

[kpb=IRP$PS_KPB]
Address of the caller's KPB (which must be a VEST KPB). KPB$PS_UCB must contain the address of a UCB and KPB$PS_IRP must contain the address of an IRP.

[idb=YES]
Flag requesting the return of the IDB address in R4. **idb**=**YES**, the default, assuming that the address of the UCB is in R5 at the time the macro is invoked, causes the address of the IDB to be placed in R4 after the channel request has been granted.

R4. If your driver does not require KP_STALL_REQCHAN to emulate the IDB returned in R4 behavior of REQCHAN (or REQPCHAN), you can save two inline MACRO-32 instructions by adding IDB=NO to the KP_STALL_REQCHAN invocation.

Description

The KP_STALL_REQCHAN macro calls IOC$KP_REQCHAN to request ownership of the controller channel. If the channel is not busy, the kernel process acquires the channel immediately and does not stall. If the channel is busy, the kernel process is placed in the channel-wait-queue to be later resumed by IOC$RELCHAN when it grants the channel request.

Only a kernel process executing at fork IPL and holding the appropriate fork lock can invoke the KP_STALL_REQCHAN macro.

KP_STALL_WFIKPCH, KP_STALL_WFIRLCH

Stall a kernel process in such a manner that it can be resumed by device interrupt processing.

Format

KP_STALL_WFIKPCH excpt ,time=65536 [,newipl=(SP)+] [,kpb=IRP$PS_KPB]

KP_STALL_WFIRLCH excpt ,time=65536 [,newipl=(SP)+] [,kpb=IRP$PS_KPB]

Parameters

excpt

Label of the timeout handling code. When the **excpt** argument is present, the macro expands to use a BLBC to transfer to that routine in the event that SS$_TIMEOUT status is returned. A driver writer may choose to omit the **excpt** argument and decode the R0 status directly.

[time=65536]

Timeout interval, expressed as the number of seconds to wait for an interrupt before a device timeout is considered to exist. A value equal to or greater than *2* is required because the timeout detection mechanism is accurate only to within one second.

[newipl=(SP)+]

IPL to which to lower before returning to 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.

[kpb=IRP$PS_KPB]

Address of the caller's KPB (which must be a VEST KPB). KPB$PS_UCB must contain the address of a UCB and KPB$PS_IRP must contain the address of an IRP.

Description

The KP_STALL_WFIKPCH and KP_STALL_WFIRLCH macros call IOC$KP_WFIKPCH and IOC$KP_WFIRLCH respectively to initiate a stall of the kernel process: These macros can only be invoked by a kernel process.

When invoked, KP_STALL_WFIKPCH or KP_STALL_WFIRLCH assumes that the local processor has obtained the appropriate synchronization with the device database by securing the appropriate device lock, as recorded in the unit control block (UCB$L_DLCK) of the device unit from which the interrupt is expected. This requirement also presumes that the local processor is executing at the device IPL associated with the lock.

KP_START

Starts the execution of a kernel process.

Format

KP_START   kpb ,routine [,registers]

Parameters

kpb
Address of KPB.

routine
Procedure value of the routine to be started as the top-level routine in the kernel process.

[registers]
Optional register save mask, indicating which registers must be preserved across kernel process context switches. Registers R0, R1, R16 through R25, R27, R28, R30, and R31 are never preserved across context switches; a **reg-mask** that indicates any of these registers is illegal. Registers R12 through R15, R26, and R29 are always saved and need not be specified.

Description

The KP_START macro calls EXE$KP_START to create a kernel process and start its execution.

When invoking the KP_START macro, code must be executing at IPL$_RESCHED or above.

KP_SWITCH_TO_KP_STACK

Switch to kernel process context.

Format

```
KP_SWITCH_TO_KP_STACK   [kpb=R6] [,return=RSB]
                        [,registers=<R0,R1,R2,R3,R4,R5,R6>]
```

Parameters

[kpb=R6]
Address of KPB.

[return=RSB]
Return semantic to be used when the kernel process stalls or completes. Valid keywords are **RSB** and **RET**.

[,registers=<R0,R1,R2,R3,R4,R5,R6>]
Register save mask, indicating which registers are to be preserved across context switches between the kernel process and the main thread.

Description

The KP_SWITCH_TO_KP_STACK macro creates a kernel process by calling EXE$KP_START, supplying a routine embedded in the macro as the top-level kernel process routine. Execution proceeds in kernel process context, with the address of the KPB in the location indicated by the **kpb** parameter and the kernel process stack active.

LOCK

Achieves synchronized access to a system resource as appropriate to the processing environment.

Format

LOCK lockname [,lockipl] [,savipl] [,condition] [,preserve=YES]

Parameters

lockname

Name of the resource to lock.

[lockipl]

Synchronization IPL. OpenVMS Alpha obtains this IPL from the spin lock data structure corresponding to the **lockname** and, thus, ignores this argument.

[savipl]

Location at which to save the current IPL.

[condition]

Indication of a special use of the macro. The only defined **condition** is **NOSETIPL**, which causes the macro to omit setting IPL.

[preserve=YES]

Indication that the macro should preserve R0 across the invocation. If you do not need to retain the contents of R0, specifying **preserve**=**NO** can enhance system performance.

Description

In a *uniprocessing* environment, the LOCK macro sets IPL to the IPL indicated by the entry in the spin lock IPL vector (SMP$AL_IPLVEC) that corresponds to the spin lock index SPL$C_**lockname**.

In a *multiprocessing* environment, the LOCK macro performs the following actions:

- Preserves R0 through the macro call (if **preserve**=**YES** is specified).
- Generates a spin lock index of the form SPL$C_**lockname** and stores it in R0.
- Calls SMP$ACQUIRE to obtain the specified spin lock. SMP$ACQUIRE indexes into the system spin lock database (a pointer to this database is located at SMP$AR_SPNLKVEC) to obtain the spin lock. Prior to securing the spin lock, SMP$ACQUIRE raises IPL to the IPL associated with the spin lock, determining the appropriate IPL from the spin lock structure (SPL$B_IPL).

In either processing environment, the LOCK macro performs the following tasks:

- Preserves the current IPL at the specified location (if **savipl** is specified)
- Sets the SMP-modified bit in the driver prologue table (DPT$V_SMPMOD in DPT$L_FLAGS)

Notes for Converting VAX Drivers

If you are converting an OpenVMS VAX driver to an Alpha driver, note that because OpenVMS Alpha obtains the synchronization IPL from the spin lock data structure corresponding to the **lockname**, it ignores the **lockipl** argument, if specified.

RELCHAN

Releases all controller data channels allocated to a device.

Format

RELCHAN

Description

The RELCHAN macro releases all controller data channels allocated to a device. When the RELCHAN macro is invoked, R5 must contain the address of the UCB. RELCHAN destroys the contents of R0 through R1.

REQCHAN

Requests exclusive use of the CRB and defines the channel grant routine entry point.

Format

REQCHAN [pri=LOW|HIGH] [,ENVIRONMENT=JSB|CALL]

Parameters

[pri=LOW]
Priority of request. If the priority is **HIGH**, REQCHAN calls IOC_STD$PRIMITIVE_REQCHANH; otherwise it calls IOC_STD$PRIMITIVE_REQCHANL.

[,environment]
Specifies the callers and grant routine environments as either JSB or CALL. The default is JSB. If specified as JSB, then an RSB is used to return from the current routine if the channel is not granted immediately and a .JSB_ENTRY directive is used to generate the grant routine. If specified as CALL, then an RET is used to return from the current routine if the channel is not granted immediately, a .CALL_ENTRY directive is used to generate the grant routine, and the grant routine parameters are copied into R3, R4, and R5.

Description

The REQCHAN macro obtains a controller's data channel.

If the channel is granted immediately, execution continues at the line of code that immediately follows the macro invocation. If no channel is available, the UCB is placed in a channel-wait queue, and the macro returns control to its caller's caller. When the channel request is granted, execution resumes at the line of code following the macro execution.

When the REQCHAN macro is invoked, R5 must contain the address of the UCB.

The REQCHAN macro returns the address of the IDB in R4 and destroys the contents of R0 through R2.

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

```
Implicit inputs:

  R3            contains a pointer to the IRP which if necessary is
                passed to the grant routine via UCB$Q_FR3(R5),

  R5            contains a pointer to the UCB,

Implicit outputs to caller:

  R4            contains the IDB address,

  R0-R2         are scratched.

Implicit outputs to grant routine, that is, entry conditions:

  ENVIRONMENT=CALL
```

A driver channel grant routine entry point is generated for a routine using the new standard call interface as described in section 4.3.

R3,R4,R5	contain traditional channel grant routine parameter values copied from the standard call interface actual parameters,
R0,R1	can be scratched.

ENVIRONMENT=JSB

A driver channel grant routine entry point is generated for a routine using the traditional JSB interface.

R3,R4,R5	contain traditional channel grant routine parameters,
R0-R5	can be scratched.

REQCOM

Places the current IRP on the post processing queue and to logically end the driver fork thread that began on entry into the start I/O or alternate start I/O routines.

Format

REQCOM [,environment=JSB | CALL]

Parameters

[,environment]
Specifies the fork routine environment as either JSB or CALL. The default is JSB. If specified as JSB, then an RSB is used to return from the current routine. If specified as CALL, then an RET is used to return from the current routine.

Description

The REQCOM macro completes the processing of an I/O request after the driver has finished its portion of the processing.

When the REQCOM macro is invoked, the following registers must contain the following values:

Register	Contents
R0	First longword of I/O status
R1	Second longword of I/O status
R5	Address of UCB

The REQCOM macro destroys the contents of R0 through R1. All other registers are also destroyed if the action of the macro initiates the processing of a waiting I/O request for the device.

REQPCHAN

Obtains a controller's data channel.

Format

REQPCHAN [pri=LOW] [,environment=JSB | CALL]

Parameters

[pri=LOW]
Priority of request. If the priority is **HIGH**, REQPCHAN calls IOC$PRIMITIVE_REQCHANH; otherwise it calls IOC$PRIMITIVE_REQCHANL.

[,environment=JSB | CALL]
Specifies the fork routine environment as either JSB or CALL. The default is JSB. If specified as JSB, then an RSB is used to return from the current routine. If specified as CALL, then an RET is used to return from the current routine.

Description

The REQPCHAN macro calls obtains a controller's data channel.

If the channel is granted immediately, execution continues at the line of code that immediately follows the macro invocation. If no channel is available, the UCB is placed in a channel-wait queue, and the macro returns control to its caller's caller. When the channel request is granted, execution resumes at the line of code following the macro execution.

When the REQPCHAN macro is invoked, R5 must contain the address of the UCB.

The REQPCHAN macro returns the address of the IDB in R4 and destroys the contents of R0 through R2.

SYSDISP

Causes a branch to a specified address according to the type of Alpha system executing the code in the macro expansion.

Format

SYSDISP list [,continue=YES]

Parameters

list
List containing one or more pairs of parameters in the following format:

<**system-type**, **destination**>

The **system-type** parameter identifies the type of Alpha system for which the macro is to generate a case table entry. The SYSDISP macro identifies the following Alpha systems:

ADU	Prototype Alpha system
DEC 4000-600	Alpha deskside system
LASER	Alpha mid-range system
DEC 3000-300	Alpha workstation
MANNEQUIN	Alpha simulator

[continue=YES]
Specifies whether execution should continue at the line immediately after the SYSDISP macro if the value at EXE$GQ_SYSTYPE does not correspond to any of the values specified as the **system-type** in the **list** argument. A fatal bugcheck of UNSUPRTCPU occurs if the dispatching code does not find the executing system identified in the **list** and the value of **continue** is NO.

Description

The SYSDISP macro provides a means for transferring control to a specified destination depending on the type of the executing system.

SYSDISP constructs appropriate symbolic constants for each **system-type** listed in **list**, and compares them against the contents of EXE$GQ_SYSTYPE. These constants have the form HWRPB$_SYSTYPE$K_*system-type*.

TBI_ALL

Invalidates the data and instruction translation buffers in their entirety.

Format

TBI_ALL [environ=MP]

Parameters

[environ=MP]
Context of translation buffer invalidation. When **environ**=**LOCAL**, the macro invalidates the translation buffer only in the context of the local processor. When **environment** is not specified or does not equal **LOCAL**, the macro extends to all system components (that is, processors and device controllers) that may have cached PTEs.

Description

The TBI_ALL macro flushes the entire contents of the data and instruction translation buffers.

The Alpha architecture specifies that whenever a PTE is modified in a way that results in a reduction of access to a virtual address, software must ensure that any cached copies of the previous PTE contents are flushed from the translation buffer before the new PTE contents can be accessed. For example, code must invalidate a translation buffer cache entry if it clears the valid bit of the associated PTE, increases its page protection, or sets one of its memory management fault bits.

If the fault-on-execute bit in the modified PTE is set, the page was never used in execution. For such pages, code can achieve better performance by avoiding an instruction translation buffer flush and invoke the TBI_DATA_64 macro to flush only the data translation buffer.

TBI_DATA_64

Invalidates a single 64-bit virtual address in the data translation buffer.

Format

TBI_DATA_64 addr [,environ=MP]

Parameters

addr

64-bit virtual address described by the translation buffer entry to be invalidated.

The TBI_DATA_64 macro assumes that the virtual address supplied in the **addr** argument will normally be in a register. Although it also accepts a memory address, you should quadword align it to avoid the performance degradation caused by the servicing of an alignment fault.

[environ=MP]

Context of translation buffer invalidation. When **environ**=**LOCAL**, the macro invalidates the translation buffer only in the context of the local processor. When **environment** is not specified or does not equal **LOCAL**, the macro extends to all system components (that is, processors and device controllers) that may have cached PTEs.

Description

As specified by the Alpha architecture, the TBI_DATA_64 macro invalidates a single 64-bit virtual address in the data translation buffer only. The instruction translation buffer is not affected.

The Alpha architecture specifies that whenever a PTE is modified in a way that results in a reduction of access to a virtual address, software must ensure that any cached copies of the previous PTE contents are flushed from the translation buffer before the new PTE contents can be accessed. For example, code must invalidate a translation buffer cache entry if it clears the valid bit of the associated PTE, increases its page protection, or sets one of its memory management fault bits.

If R2 is not specified as the **addr** argument, it is preserved across the macro call.

The TBI_DATA_64 macro and its callers depend on the MTPR instruction to save and restore the registers it destroys. If a MACRO compiler built-in is ever used again, those registers must be specifically preserved.

TBI_SINGLE

Flushes the cached contents of a single page-table entry (PTE) from the data and instruction translation buffers.

Format

TBI_SINGLE addr [,environ=MP]

Parameters

addr
32-bit virtual address to be invalidated.

[environ=MP]
Context of translation buffer invalidation. When **environ**=**LOCAL**, the macro invalidates the translation buffer only in the context of the local processor. When **environment** is not specified or does not equal **LOCAL**, the macro extends to all system components (that is, processors and device controllers) that may have cached PTEs.

Description

As specified by the Alpha architecture, the TBI_SINGLE macro flushes the cached contents of a single page-table entry (PTE) from both the data and instruction translation buffers.

The Alpha architecture specifies that whenever a PTE is modified in a way that results in a reduction of access to a virtual address, software must ensure that any cached copies of the previous PTE contents are flushed from the translation buffer before the new PTE contents can be accessed. For example, code must invalidate a translation buffer cache entry if it clears the valid bit of the associated PTE, increases its page protection, or sets one of its memory management fault bits.

If the fault-on-execute bit in the modified PTE is set, the page was never used in execution. For such pages, code can achieve better performance by avoiding an instruction translation buffer flush and invoke the TBI_DATA_64 macro to flush only the data translation buffer.

TBI_SINGLE_64

Invalidates a single 64-bit virtual address in both the data and instruction translation buffers.

Format

TBI_SINGLE_64 addr [,environ=MP]

Parameters

addr

64-bit virtual address to be invalidated.

The TBI_SINGLE_64 macro assumes that the virtual address supplied in the **addr** argument will normally be in a register. Although the TBI_DATA_64 macro also accepts a memory address, you should quadword align it to avoid the performance degradation caused by the servicing of an alignment fault.

[environ=MP]

Context of translation buffer invalidation. When **environ**=**LOCAL**, the macro invalidates the translation buffer only in the context of the local processor. When **environment** is not specified or does not equal **LOCAL**, the macro extends to all system components (that is, processors and device controllers) that may have cached PTEs.

Description

As specified by the Alpha architecture, the TBI_SINGLE_64 macro invalidates a single 64-bit virtual address in both the data and instruction translation buffers.

The Alpha architecture specifies that whenever a PTE is modified in a way that results in a reduction of access to a virtual address, software must ensure that any cached copies of the previous PTE contents are flushed from the translation buffer before the new PTE contents can be accessed. For example, code must invalidate a translation buffer cache entry if it clears the valid bit of the associated PTE, increases its page protection, or sets one of its memory management fault bits.

If R2 is not specified as the **addr** argument, it is preserved across the macro call.

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.

Format

```
TIMEDWAIT   time [,ins1] [,ins2] [,ins3] [,ins4] [,ins5] [,ins6] [,donelbl] [,imbedlbl]
            [,ublbl] [,nsec] [,bus] [,userins]
```

Parameters

time

Delay time specified in 10-microsecond intervals. Actual delay time depends on number and type of loop instructions, clock frequency, and other variables.

Note that the **time** and **nsec** arguments are mutually exclusive.

[ins1]

First instruction to be executed in the delay loop.

[ins2]

Second instruction to be executed in the delay loop.

[ins3]

Third instruction to be executed in the delay loop.

[ins4]

Fourth instruction to be executed in the delay loop.

[ins5]

Fifth instruction to be executed in the delay loop.

[ins6]

Sixth instruction to be executed in the delay loop.

[donelbl]

Label placed after the instruction at the end of the TIMEDWAIT loop; embedded instructions can pass control to this label in order to pass control to the instruction following the invocation of the TIMEDWAIT macro.

[imbedlbl]

Label placed at the first of the embedded instructions; after executing a processor-specific delay, the TIMEDWAIT macro passes control here to retest for the condition.

[ublbl]

Label placed at the instruction that performs the processor-specific delay after each execution of the loop of embedded instructions; embedded instructions can pass control here in order to skip the execution of the rest of the embedded instructions in a given execution of the embedded loop.

[nsec]

Delay time in nanoseconds. Actual delay time depends on number and type of loop instructions, clock frequency, and other variables.

Note that the **nsec** and **time** arguments are mutually exclusive.

[bus]
Address of ADP of the bus, if a bus-specific delay should be added to the delay loop. You would add a bus-specific delay, for instance, to avoid saturating a bus with CSR references in the instruction loop.

[userins]
Additional instructions to be executed in the delay loop. This list can be of indefinite length and is executed after (or in place of) the instructions specified in the **ins1** through **ins6** arguments.

Description

The TIMEDWAIT macro provides the ability to write code that is based on specific time intervals and is independent of system-specific timer implementations. You can use the TIMEDWAIT macro for the following tasks:

- Timeout handling. The macro generates a time delay in which a number of instructions tests for the occurence of a specific event or condition. In this case, either the specified time delay is completed or an exit condition is met. A device driver uses this mechanism to establish time bounds for the execution of a given instruction sequence.
- Simple delay. The macro generates a time delay with no embedded instructions. When the delay has completed, the code thread continues.
- Optimistic polling. If a device is known to respond quickly, a driver might invoke the TIMEDWAIT macro with a short time delay to check for device completion and potentially avoid the overhead of device interrupt servicing. If the instructions supplied to the delay loop determine that the operation has completed, an interrupt has been saved. Otherwise, the delay completes and the suspended code thread continues.

The TIMEDWAIT macro returns a status code (SS$_NORMAL or SS$_TIMEOUT) in R0. Note that the embedded instructions can overwrite SS$_NORMAL status, although SS$_TIMEOUT status cannot be overwritten. The macro destroys the contents of R1, and preserves all other registers.

Examples

1.

```
TIMEDWAIT TIME=#600*1000,-         ;6-second wait loop
          INS1=<TSTB  RL_CS(R4)>,- ;Is controller ready?
          INS2=<BLSS  15$>,-       ;If LSS - yes
          DONELBL=15$              ;Label to exit wait loop
BLBC      R0,25$                   ;Time expired - exit
```

2.

```
TIMEDWAIT NSEC=#<100*1000>,-
          DONELBL=10$,-            ;Label to exit wait loop
          USERINS=<<BITB #1,CSR(R4)>,<BNEQ 10$>> ;Check CSR
```

Notes for Converting VAX Drivers

If you are converting an OpenVMS VAX driver to an Alpha driver, note the following:

- The OpenVMS Alpha TIMEDWAIT macro, unlike the OpenVMS VAX version, *does* read a processor register. As such, the interval it waits corresponds very closely with the delay specified in the **time** or **nsec** argument, and is not affected by the number or complexity of the imbedded instructions that may be specified as arguments.
- The OpenVMS Alpha TIMEDWAIT macro does not automatically adjust the **time** (or **nsec**) argument to accommodate a bus-specific or CPU-specific delay factor. If a bus-specific delay is needed, you can request one by specifying the address of the bus's ADP in the **bus**.
- The OpenVMS Alpha TIMEDWAIT macro allows you to specify the delay time in either 10-microsecond intervals (using the **time** argument) or in nanosecond units (using the **nsec** argument).
- The OpenVMS Alpha TIMEDWAIT macro allows you to specify, in the **userins** argument, instructions to execute within the delay loop and test for exit conditions. These instructions execute within the loop after (or in place of) the instructions specified in the **ins1** through **ins6** arguments.

WFIKPCH, WFIRLCH

Suspends a driver fork thread and folds its context into a fork block in anticipation of a device interrupt or timeout. When WFIKPCH is invoked, the fork thread keeps ownership of the controller channel while waiting; when WFIRLCH is invoked, the fork thread releases ownership of the controller channel.

Format

$\left\{ \begin{array}{l} \text{WFIKPCH} \\ \text{WFIRLCH} \end{array} \right\}$ excpt [,time=65536] [,newipl] [environment=JSB | CALL] [,toutrout]

Parameters

excpt
Label of the timeout handling code within the driver.

[time=65536]
Timeout interval, expressed as the number of seconds to wait for an interrupt before a device timeout is considered to exist. A value equal to or greater than *2* is required because the timeout detection mechanism is accurate only to within 1 second.

[newipl=(SP)+]
IPL to which to lower before returning to 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.

[,environment]
Specifies the current and interrupt resume routine environments are either JSB or CALL. The default is JSB. If specified as JSB, then the return from the current procedure is via a RSB, and a .JSB_ENTRY directive is used to generate the resume routine entry point. If specified as CALL, then the return from the current procedure is via a RET, a .CALL_ENTRY directive is used to generate the resume routine entry point, and the IRP, FR4, and UCB parameters in the resume routine are copied into R3, R4, and R5.

[,toutrout]
Specifies the timeout routine entry point. The timeout routine is a fork routine that can either use the traditional or the new standard call interface. If not specified then the resume routine entry point is also used as the timeout routine entry point. The timeout routine procedure value is loaded into UCB$PS_TOUTROUT(R5) for potential use by EXE$TIMEOUT. This parameter cannot be specified together with the EXCPT parameter.

Description

The WFIKPCH and WFIRLCH macros construct an inline entry point for the code that follows the macro invocation (normally called the fork routine). They 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 (specified in the **excpt** argument) if it is set.

Finally, WFIKPCH and WFIRLCH place the procedure value of the fork routine (at the instruction following the macro invocation) in UCB$L_FPC, insert the **time** value in R1 and **newipl** value in R2, and call the appropriate wait-for-interrupt routine (either IOC$PRIMITIVE_WFIKPCH or IOC$PRIMITIVE_WFIRLCH).

When the wait-for-interrupt routine returns control, the WFIKPCH or WFIRLCH macro issues an RSB instruction to the caller of the routine which invoked it (that is, the caller of the start-I/O routine).

Either the device interrupt servicing routine or the software timer interrupt servicing routine will eventually issue a JSB instruction to the fork routine. In both instances, code can assume that only R3 and R4 have been preserved across the suspension.

IOC$WFIKPCH and IOC$WFIRLCH assume that, prior to the invocation of the macro, a DEVICELOCK macro has been issued to synchronize with other device activity.

When the WFIKPCH or WFIRLCH macro is invoked, the following locations must contain the values listed:

Location	**Contents**
R5	Address of UCB
00(SP)	IPL at which control is passed to the caller's caller (if the **newipl** argument is not specified)

Notes for Converting VAX Drivers

If you are converting a VAX driver to an Alpha driver, note the following:

```
Implicit inputs:

  R3          contains a pointer to the IRP which  is  passed  to
              the interrupt resume routine via UCB$Q_FR3(R5),

  R4          contains the 64-bit value to pass to the  interrupt
              resume routine via UCB$Q_FR4(R5),

  R5          contains a pointer to the UCB,

Implicit outputs to caller:

  R0,R1,R2    are scratched.

Implicit outputs to resume routine, i.e. entry conditions:

  ENVIRONMENT=CALL

              An entry point is generated that conforms  to  both
              the new standard call interface for a driver resume
              from interrupt routine as described in section  4.7
              and  the  new  standard  call  interface for a fork
              routine as described in section 4.2.

    R3,R4,R5  contain traditional resume from  interrupt  routine
              parameter  values (64-bit value for R4) copied from
              the standard call interface actual parameters,

    R0,R1     can be scratched.

  ENVIRONMENT=JSB
```

An entry point is generated that conforms to both the traditional JSB interface for a driver resume from interrupt routine
and the traditional JSB interface for a fork routine..

R3,R4,R5	contain traditional resume from interrupt routine parameters (64-bit value for R4),
R0-R4	can be scratched.

Index