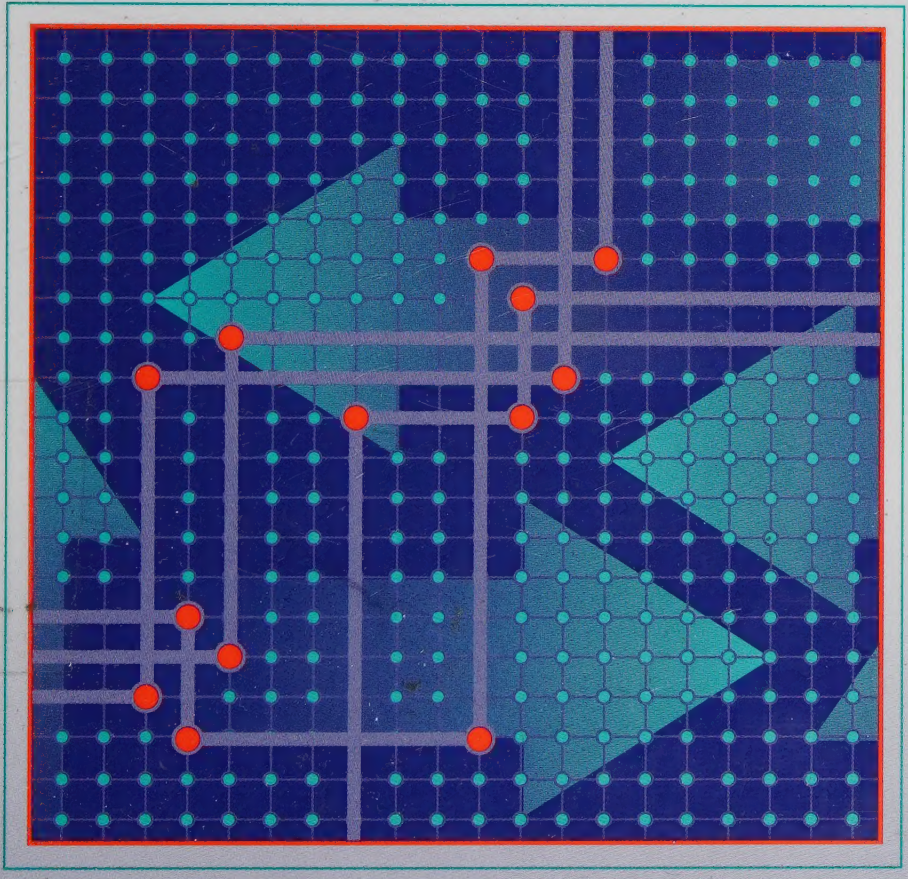
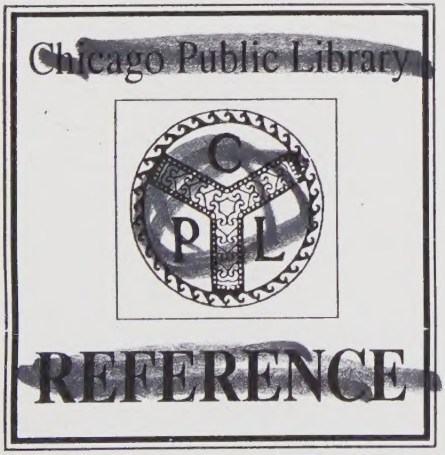
Writing OpenVMS Alpha Device Drivers in C

DEVELOPER'S GUIDE AND REFERENCE MANUAL



Margie Sherlock • Leonard Szubowicz

Writing OpenVMS Alpha Device Drivers in C

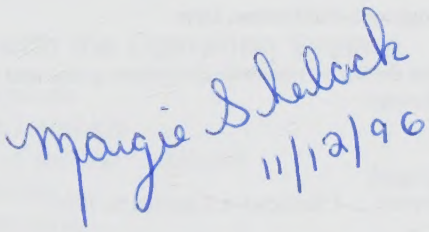


Digital Press Editorial Board

Writing OpenVMS Alpha Device Drivers in C

Developer's Guide and Reference Manual

Margie Sherlock
Leonard S. Szubowicz



Digital Press
Boston Oxford Johannesburg Melbourne New Delhi Singapore

Digital Press™ is an imprint of Butterworth–Heinemann, Publisher for Digital Equipment Corporation.

A member of the Reed Elsevier group

Recognizing the importance of preserving what has been written, Butterworth–Heinemann prints its books on acid-free paper whenever possible.

The document was prepared using VAX DOCUMENT Version 2.1

Library of Congress Cataloging-in-Publication Data

Sherlock, Margie, 1957–
Writing OpenVMS alpha device drivers in C: developers guide and reference manual / Margie Sherlock, Leonard S. Szubowicz.
p. cm.
Includes index.
ISBN 1–55558–133–1 (paper)
1. OpenVMS device drivers. I. Szubowicz, Leonard S., 1954–
II. Title.
QA76.9.D49S54 1996
005.7'1265--dc20 95–47199
CIP

British Library Cataloguing-in-Publication Data
A catalogue record of this book is available from the British Library.

The publisher offers special discounts on bulk orders of this book.
For information, please contact:
Managers of Special Sales
Butterworth–Heinemann
313 Washington Street
Newton, MA 02158-1626
Tel: 617-928-2500
Fax: 617-928-2620

For information on all Digital Press publications available, contact our World Wide Web home page at http://www.bh.com/dp

Order number: EY-T133E-DP

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Contents

3 Synchronization of I/O Request Processing

Part II Creating OpenVMS Alpha Device Drivers

4 Accessing Device Interface Registers

5 Allocating Map Registers

6 Writing FDT Routines

7 Writing a Start-I/O Routine

8 Using the Simple Fork Start-I/O Mechanism

Part III Running OpenVMS Alpha Device Drivers

11 Compiling and Linking a Device Driver

12 Loading a Device Driver

13 Debugging a Device Driver

Part IV Bus Support Information

14 PCI Bus Support

15 ISA Bus Support

16 EISA Bus Support

Part V Reference

17 Data Structures

18 Device Driver Entry Points

19 System Routines

20 C Driver Macros

Figures

Tables

Preface

If you want to write an OpenVMS Alpha device driver in the C programming language, the information you need is right here. This book identifies the components of OpenVMS Alpha device drivers, explains their role in the operating system, describes how to code drivers, and contains plenty of live code examples.

The book contains six parts, each of which covers different aspects of OpenVMS Alpha device driver development:

- Part 1 describes the components of OpenVMS Alpha device drivers and discusses how drivers interact with the operating system.
- Part 2 describes what you need to know to code each part of an OpenVMS Alpha device driver.
- Part 3 describes how to compile, link, load, and debug an OpenVMS Alpha device driver.
- Part 4 describes bus-specific and processor-specific details that affect OpenVMS Alpha device drivers.
- Part 5 provides reference material about the routines and macros used in OpenVMS Alpha device drivers.
- Part 6 contains OpenVMS Alpha System address maps, a sample IOGEN Configuration Building Module (ICBM), and a sample driver written in C.

Each part of the book begins with a page that summarizes its contents.

Assumptions

This book assumes that you are familiar with programming for the OpenVMS operating system and that you understand some basic device driver concepts. It is not an introductory programming book or a device driver writing tutorial.

Specifically, the book is designed for systems engineers who:

- Understand OpenVMS operating system and programming concepts.
- Are experienced with programming in C.
- Understand the hardware devices for which their drivers are being written.

Although this book is intended for system engineers who want to write device drivers, anyone who wants to learn about OpenVMS Alpha device drivers will also find the book useful.

Getting More Information

If you need additional information about OpenVMS Alpha internals, the Alpha Architecture, or OpenVMS Alpha operating system concepts, refer to the following books:

- *OpenVMS Alpha Internals and Data Structures*

  Ruth E. Goldenberg and Saro Saravanan
  (Available from Digital Press)

- *Alpha AXP Architecture Reference Manual*

  Richard L. Sites and Richard T. Witek
  (Available from Digital Press)

- Specific manuals in the OpenVMS Documentation Set

Send Us Your Comments

We welcome your comments about this book. If you have suggestions for improving a particular section or find any errors, please indicate the chapter, section, and page number. We also welcome more general comments (and compliments, too!).

Please address all correspondence to our email address:

`vms_drivers@zko.dec.com`

We look forward to hearing from you!

Conventions

This book uses the following conventions:

Every use of OpenVMS Alpha means the OpenVMS Alpha operating system. The name of the OpenVMS AXP operating system has been changed to OpenVMS Alpha. Any references to OpenVMS AXP or AXP in this book are synonymous with OpenVMS Alpha or Alpha.

The following conventions are also used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	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 surround a required choice of options; you must choose one of the options listed.

boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason (user action that triggers a callback). Boldface text is also used to show user input in Bookreader versions of the manual.
italic text	Italic text emphasizes important information and indicates complete titles of manuals.
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a function code, the name of a file, or the abbreviation for a system privilege.
monospace type	Monospace type in text identifies C macros, data structure member names, and header files. Information in this font appears exactly as you would specify it in a C program. A hyphen in code examples indicates that additional arguments to the request are provided on the line that follows.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Acknowledgments

Many people contributed information to this book, reviewed countless drafts, and extensively supported our efforts. Their talents and expertise, interest and enthusiasm made this book worth writing. We gratefully acknowledge and sincerely thank the following:

The OpenVMS Alpha High-Level Language Device Driver engineering team who designed and developed the device driver interfaces we describe and who reviewed many sections of the book: Walter Arbo, Walter Blaschuk, Steve Dipirro, David Eiche, Anne McElearney.

Managers who provided initial and ongoing support for the project: Joan Winslow, Steve Noyes, Pat St. Laurent, Ken Munsell, Martin King, Howard Hayakawa.

John Croll, for providing the ICBM information and examples, reading the entire book, using it to write a driver, and for making hundreds of suggestions that greatly improved the technical accuracy and completeness of this book.

Cathy Fariz, for providing valuable comments, contributing many code examples, and whose enthusiasm and good humor often kept the book on track.

Reviewers who made numerous, important technical contributions and provided code examples: Jim Janetos, Ruth Goldenberg, Forrest Kenny, Sue Lewis, Fred Kleinsorge, Scott Apgar, Reuven Somberg, Mark Jilson, Greg Rogers.

Denise Dumas, for revising and updating the EISA bus chapter.

Reviewers who read particular chapters: Burns Fisher, Nitin Karkhanis, Drew Mason, Richard Bishop, Richard Sayde.

People who provided a tremendous amount of support and encouragement: Ben Thomas, Richard Critz, Ralph Weber, Karen Noel, Steve Zalewski, Carolyn Wurm, Mike Meagher, Jim Parker, Susan Rist.

Editing, production, and art support: Merle Roesler, Pat Walker, Natalie Pitula, Brenda Vezina, Marge Shiel.

Liz McCarthy, from Digital Press, for all of her help with this project.

Special Thanks

We especially thank our families.

Terry Sherlock, for everything...

Leora and Sarah Szubowicz, for the support and joy beyond fair measure...

Part I

Overview of OpenVMS Alpha Device Drivers

Part I describes the components of OpenVMS Alpha device drivers and discusses their role in the operating system. It includes the following chapters:

- Chapter 1 provides an introduction to OpenVMS Alpha devices drivers and contains information you should read before you start writing your own driver.
- Chapter 2 explains how device drivers interact with the operating system to handle I/O requests, and it summarizes driver components and activities.
- Chapter 3 discusses synchronization concepts most important to device-driver writers.

1

Introduction

A **device driver** is a set of routines and tables that an operating system uses to process an I/O request for a particular device such as a disk, tape, or network controller. Each type of device has a separate device driver (or simply, a driver), which is primarily responsible for communicating system and user I/O requests to I/O hardware and ensuring that they are carried out correctly within a resonable time.

A system utility loads a device driver into system virtual address space and creates its associated data structures. Once loaded, a device driver controls I/O operations on a peripheral device by performing the following functions:

- Defining the peripheral device for the rest of the operating system
- Preparing a device unit and its controller (or both) for operation at system startup and during recovery from a power failure
- Performing device-dependent I/O preprocessing
- Translating programmed requests for I/O operations into device-specific commands
- Activating a device unit
- Responding to hardware interrupts generated by a device unit
- Responding to device timeout conditions
- Responding to requests to cancel I/O on a device unit
- Reporting device errors to an error-logging program
- Returning status from a device unit to the process that requested the I/O operation

1.1 Types of OpenVMS Alpha Drivers

The OpenVMS Alpha operating system supports a variety of peripheral disk and tape devices, as well as terminals, networks, and mailboxes (virtual devices for interprocess communication).

These peripherals are supported by two types of OpenVMS Alpha driver models: the **traditional model**, in which the entire model is contained in one image, and the **class/port model**, in which the driver consists of more than one image.

In the class/port model, the class driver performs functions common to a class of device, such as an MSCP disk or a Small Computer System Interface (SCSI) tape, for example. The port driver contains controller-specific subroutines for the class driver. For any given device, a class driver is bound to a port driver through the unit control block (UCB) or a port-specific data structure. The functional division into class and port drivers simplifies maintenance and integration of software when new devices and controllers are introduced.

The following types of devices use the class/port interface.

- MSCP devices
- SCSI devices
- Terminal devices
- Communication devices

1.2 Writing Your Own OpenVMS Alpha Driver

Digital supplies drivers for all devices supported by the OpenVMS Alpha operating system and provides system service routines to access the special device-dependent features available in many of these devices. User's, however, can write their own device drivers to attach non-Digital supplied devices to OpenVMS Alpha systems.

Before you write a device driver, you should look at an existing driver that supports a device similar to the one you plan to support, if one is available. If you have access to the OpenVMS Alpha source listings kit, look at the drivers in the listing directory. If you do not have access to this kit, look at the sample driver in Appendix B.

If you are creating your own driver, you can design your own interfaces. The interface between the drivers (for example, port and class) is totally up to the driver designer. If you are writing a driver to work with an existing driver (for example, a SCSI port driver to work with an existing SCSI class driver), you must follow the rules for the existing driver.

1.3 Benefits of Writing OpenVMS Alpha Drivers in C

The main benefit of using C to write a device driver is that C is easier to write than assembly-level code.

Coding device drivers in C also provides a certain level of portability. The ability to write portable code is especially important in multiplatform and "multi-operating system" development environments. By writing OpenVMS Alpha device drivers in C, drivers are easier to modify for different platforms running OpenVMS Alpha, and drivers are migrated more easily to OpenVMS Alpha systems.

Finally, for RISC machines (like Alpha), code generated by a good optimizing C compiler is usually much more efficient than code produced using the MACRO-32 language for Alpha.

1.4 OpenVMS Features for Writing Drivers in C

The OpenVMS Alpha operating system provides many features that facilitate developing device drivers in C. These include the following:

- Standard call interfaces to existing operating system routines
- New operating system routines
- Call interfaces for device driver table-building macros
- Numerous C macros

These interfaces are included in the SYS$LIBRARY:SYS$LIB_C.TLB library. Specific details about accessing header files for system routines, macros, and data structures are included throughout this book.

2

How Drivers Interact with the Operating System

This chapter presents an an overview of how device drivers interact with the operating system to handle I/O requests, and it summarizes driver components and activities. This chapter also provides references to other chapters that contain more details about the topics introduced here.

2.1 Handling I/O Requests

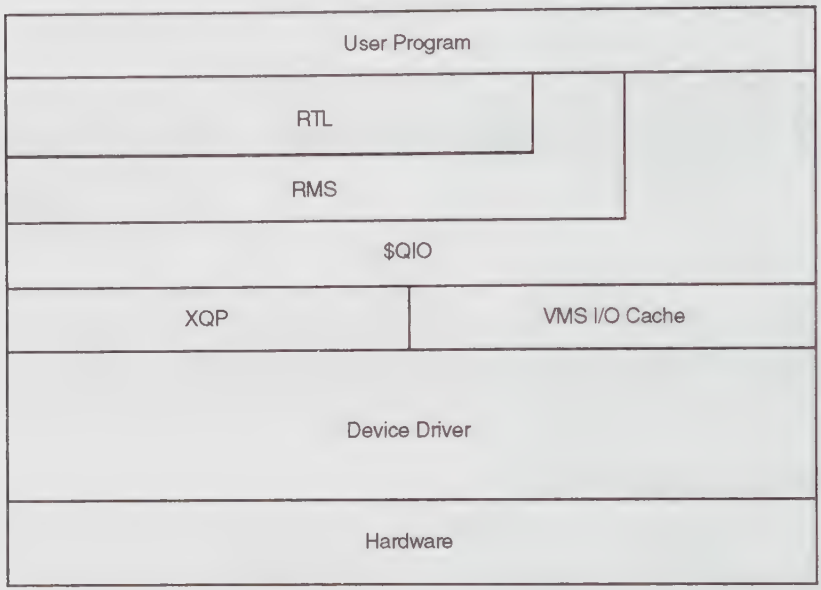
User images initiate most OpenVMS I/O operations. The OpenVMS executive initiates I/O operations for swapping, paging, file system, and other miscellaneous functions.

User images and most OpenVMS components request an I/O operation through the $QIO service, which provides a device-independent user interface to drivers. If the device-independent parameters of a user's I/O request are valid, $QIO allocates and builds an I/O request packet (IRP).

The OpenVMS I/O system is composed of several layers, as shown in Figure 2–1. Each layer provides a higher-level interface to the level about it. At the highest level are the record management and data base systems that manage named objects such as files, records, fields, and so on. At the lower levels are the kernel routines that interface directly to physical devices.

A process (identified by a PCB) obtains a "channel" to a named device. The process uses the $QIO service to issue an I/O request on a channel. The $QIO service specifies a function code and up to 6 function specific parameters. There are 64 function codes, but each function code can support a set of function modifiers. For example, "no echo" and "data check" are function modifiers on the "read" function.

Figure 2–1 Layers of I/O



ZK-7508A-GE

2.2 A Tour of the I/O Database

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 in nonpaged pool collectively known as the **I/O database**.

The I/O database serves the following main functions:

- It describes individual I/O hardware components, such as devices, controllers, and adapters, as well as device drivers.
- It describes the configuration of and relationships between I/O hardware components.
- It maintains the context of I/O operations.

2.2.1 Components of the I/O Database

The I/O database consists of the following three parts:

- Driver tables that allow the system to load drivers, to validate device functions, and to call driver routines at their entry points
- Data structures that describe I/O bus adapters, device types, device units, device controllers, and logical paths from processes to devices
- I/O request packets that define individual requests for I/O activity

The following sections briefly describe these components, and Section 2.2.2 shows their relatationships to each other.

2.2.1.1 Driver Tables

Three driver tables are defined in every driver: driver prologue table, driver dispatch table, and function decision table.

The **driver prologue table** (DPT) defines the identity and attributes or characteristics of the driver to the system utility that loads the driver into virtual memory and creates the associated data structures. With the information provided in the DPT, the driver-loading procedure can load drivers and perform the necessary I/O database initialization.

Section 10.3 describes how to create a DPT and further discusses its functions.

The **driver dispatch table** (DDT) lists the addresses of the entry points of standard routines within the driver, and identifies the size of the diagnostic and error message buffers for drivers that perform error logging. Section 10.4 contains additional information about DDTs. The structure and contents of the DDT are described in Chapter 17.

The **function decision table** (FDT) lists the addresses of the appropriate I/O preprocessing routine, called an **upper-level FDT action routine**, for each of the 64 $QIO function codes. The FDT also identifies which functions are "buffered" and which are "direct". (For more information about buffered and direct I/O, see Section 2.7.) A device driver contains device-dependent FDT routines, and the operating system provides routines (described in Section 6.4.1) that perform request preprocessing common to many I/O functions.

When a user process calls the $QIO system service, the system service uses the I/O function code specified in the request to select the appropriate upper-level FDT action routine. To prepare for the actual I/O operation, FDT routines perform such tasks as allocating buffers in system space, locking pages in memory, and validating the device-dependent arguments (**p1** to **p6**) of the

$QIO request. For more information about FDT routines, see Chapter 6 and Chapter 10.

2.2.1.2 Data Structures

I/O database data structures describe peripheral hardware and are used by the operating system to synchronize access to devices. The operating system creates these data structures either at system startup or when a driver is loaded into the system.

This section quickly reviews many of the primary data structures used by OpenVMS device drivers. Detailed descriptions of I/O database structures and their fields appear in Chapter 17.

The system defines a **unit control block** (UCB) for each device unit attached to the system. A UCB defines the characteristics and current state of an individual device unit.

UCBs are the focal point of the I/O database. When a driver is suspended or interrupted, the UCB keeps the context of the driver in a set of fields collectively known as a **fork block**.

Note that other structures, such as the CRB, also include a fork block. The discussion of fork blocks and fork processes in Section 2.5 explains the role of fork blocks in driver processing.) In addition, the UCB contains the listhead for the queue of pending I/O request packets (IRPs) for the unit.

A **device data block** (DDB) contains information common to all devices of the same type that are connected to a particular controller. It records the generic device name concatenated with the controller designator (for example, LPA, DKB), and the name and location of the associated device driver. In addition, the DDB contains a pointer to the first UCB for the device units attached to the controller.

The operating system creates a **channel request block** (CRB) for each controller. A CRB defines the current state of the controller and lists the devices waiting for the controller's data channel. It also contains a pointer to the **interrupt service routine** (ISR).

The system also creates an **interrupt dispatch block** (IDB) for each controller. An IDB lists the device units associated with a controller and points to the UCB of the device unit that the controller is currently servicing. In addition, an IDB points to device registers and the controller's I/O adapter.

An **adapter control block** (ADP) defines the characteristics and current state of an I/O adapter such as the TURBOchannel interface on a DEC 3000. An ADP contains the information necessary to allocate the adapter's resources. The operating system provides routines that drivers can call to interface with the appropriate adapter.

The **channel control block** (CCB) describes the logical path between a process and the UCB of a specific device unit. Each process owns a number of CCBs. When a process issues the Assign I/O Channel ($ASSIGN) system service, the system writes a description of the assigned device to the CCB.

Note that channel request blocks (CRBs) and channel control blocks (CCBs) are two separate data structures. To distinguish the two, it may be helpful to think of the channel request block as the "controller request" block because it describes the hardware controller. In contrast, the channel control block is used by a process and a device unit to manage the logical path (the `channel` parameter to the $ASSIGN and $QIO system services) in performing I/O operations.

Unlike the data structures mentioned earlier, a CCB is not located in shared system address space, but in the process-private address space.

2.2.1.3 I/O Request Packets

The third part of the I/O database is a set of I/O request packets. When a process requests I/O activity, the operating system constructs an **I/O request packet** (IRP) that describes the I/O request in a standard form.

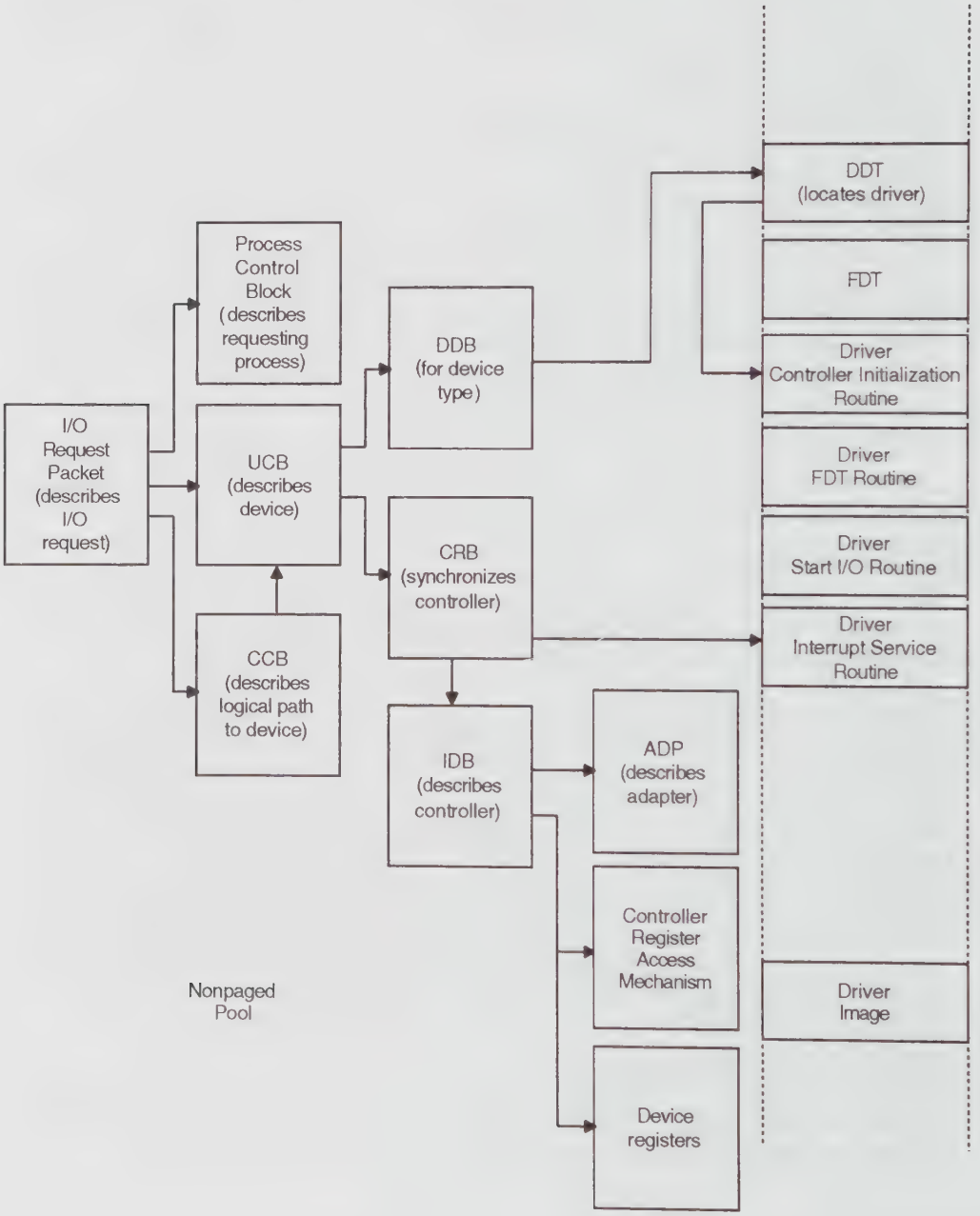
The IRP contains fields into which the system and driver I/O preprocessing routines can write information: for instance, the device-dependent arguments specified in the call to the $QIO system service. The packet also includes buffer addresses, a pointer to the target device's UCB, an I/O function code, and pointers to the I/O database. After preprocessing, the IRP can be queued to a list originating in the device's UCB to await processing by the driver.

When the device unit is free and the IRP is next in line to be processed on the unit, the system sends it to the device driver's start-I/O routine. The start-I/O routine uses the IRP as its source of detailed instructions about the operation to be performed. (For more information about start-I/O routines, see Section 2.4.3.)

2.2.2 Snapshot of the I/O Database

Figure 2–2 illustrates some of the relationships between system I/O routines, the I/O database, and a device driver.

Figure 2–2 Overview of the I/O Database



2.3 Synchronization of Driver Activity

It is important to know that a device driver does not run sequentially from beginning to end. Rather, the operating system uses driver tables and other information maintained by itself and the driver to determine which driver routines to activate and when they should be activated.

Device drivers and other kernel-mode code must maintain synchronization with other priority operating system activities. The term **synchronization** refers to the means by which such code accesses shared data in a consistent, orderly, and predictable way. Because there may be more than one processor active in an OpenVMS Alpha system, system-level code must synchronize its actions with other code threads it may have preempted on the same (or *local*) processor, as well as with those that are active (or to be activated) on other processors in the system.

This section briefly describes OpenVMS synchronization concepts. For more information about synchronization techniques, see Chapter 3.

The operating system uses hardware and software **interrupt priority levels (IPLs)** to order system events on each local processor in an OpenVMS Alpha system. OpenVMS Alpha hardware defines 32 interrupt priority levels (IPLs). The higher numbered IPLs (16 to 31) are reserved for hardware interrupts, such as those posted by devices. The operating system uses the lower numbered IPLs (0 to 15). Code that executes at a higher IPL takes precedence over code that executes at a lower IPL. A driver, in conjunction with the operating system, ensures that it maintains system synchronization by performing certain activities and by accessing certain data only at the appropriate IPL.

In a multiprocessing system, the driver extends the synchronization it achieves by executing locally at a given IPL by acquiring ownership of the **spinlock** associated with the operation it is performing. A spinlock is a semaphore associated with a set of system structures, fields, or registers whose integrity is critical to the performance of a specific operating system task. For more information about IPLs, spinlocks, and other forms of synchronization, see Chapter 3.

2.4 Overview of Driver Routines

As mentioned in the beginning of this chapter, device drivers contain tables and routines that enable drivers to process I/O requests. This section briefly describes many of the routines used by OpenVMS Alpha drivers. For more details about each type of routine, see the appropriate chapter.

2.4.1 Initialization Routines

Initialization routines are invoked when loading your driver and configuring the I/O database structures that are used by your driver. This section defines the routines used in the driver initialization sequence. For an overview of driver initialization and more details about using the routines, see Chapter 10.

The **driver table initialization routine** completes the initialization of the DPT, DDT, and FDT structures. If a driver image contains a routine named DRIVER$INIT_TABLES, this routine is called once by the driver loading mechanism immediately after the driver image is loaded and before any validity checks are performed on the DPT, DDT, and FDT.

A **structure initialization routine** is called once for each unit by the driver loading mechanism after that UCB is created. At the point of this call, the UCB has not yet been fully linked into the I/O database. This routine is responsible for filling in driver specific fields in the I/O database structures that are passed as parameters to this routine. Then the driver loading mechanism completes the integration of this device specific structure into the I/O database. Note that this routine must confine its actions to filling in these I/O database structures and may not attempt to initialize the hardware device. Initialization of the hardware device is the responsibility of the controller and unit initialization routines which are called some time later.

Before you access device registers on Alpha systems, you must map the device registers into the processor's virtual address space. The **platform independent I/O bus mapping routine** IOC$MAP_IO maps I/O bus physical address space into an address region accessible by the processor. Once your device is mapped, you can access it using the IOC$READ_IO and IOC$WRITE_IO routines or the CRAM data structure and associated routines. For more information about the mapping and access routines available for accessing device registers, see Chapter 4.

The **unit initialization routine** and **controller initialization routine** prepare a device or controller for operation when the driver-loading procedure loads the driver into memory and when the system recovers from a power failure. The amount and type of initialization needed by devices and controllers vary according to the device type and the I/O bus to which the device or controller is attached.

2.4.2 FDT Routines

FDT routines perform driver-specific I/O processing while still in the context of the $QIO system service caller. Driver I/O function preprocessing on OpenVMS Alpha systems often requires the cooperative efforts of upper-level FDT action routines, FDT support routines, and FDT completion routines.

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 `ini_fdt_act macro` in the DRIVER$INIT_TABLES routine. FDT dispatching code in the $QIO system service calls an upper-level FDT action routine, passing to it the addresses of the I/O request packet (IRP), process control block (PCB), unit control block (UCB), and channel control block (CCB). An upper-level FDT action routine must return SS$_FDT_COMPL status to the $QIO system service. (See Chapter 19 for a full description of the formal interface to an upper-level FDT action routine.)

OpenVMS provides a set of upper-level FDT action routines, and drivers can also define their own driver-specific upper-level FDT action routines.

FDT completion routines terminate FDT processing, forward an IRP to its next destination, and return back to their callers.

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 provides a set of FDT support routines, and drivers can also include their own support routines. EXE_STD$READCHK is an example of an FDT support routine.

A **composite** FDT routine is an upper-level action routine that is required when a single I/O function code must be processed by more than one upper-level FDT routine. OpenVMS Alpha 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).

2.4.3 Start-I/O Code Path Routines

The Start-I/O to request complete code path services I/O requests by interacting with the device controller. This code path can use either the simple fork or kernel process mechanisms. Because each mechanism uses a different set of routines, the routines your start-I/O code path uses depends on which mechanism you choose. For more information about selecting the start-I/O mechanism most suitable for your driver, see Chapter 7.

Start-I/O routines performs such additional device-dependent tasks as translating the I/O function code into a device-specific command, storing the details of the user request in the device's UCB in the I/O database and, if necessary, obtaining access to controller and adapter resources. Whenever the start-I/O routine must wait for these resources to become available, the operating system suspends the routine, reactivating it when the resources become free.

The start-I/O routine ultimately activates the device by suitably loading the device's registers. At this stage, the start-I/O routine invokes a system macro that causes its execution to be suspended until the device completes the I/O operation and posts an interrupt to the processor. The start-I/O routine remains suspended until the driver's **interrupt service routine** handles the interrupt.

When a device posts an interrupt, its driver's interrupt service routine determines whether the interrupt is expected or unexpected, and takes appropriate action. If the interrupt is expected, the interrupt service routine reactivates the driver's start-I/O routine at the point of suspension. The general course of action of driver mainline code at this time is to perform device-dependent I/O postprocessing and to transfer control to the operating system for device-independent I/O postprocessing.

A description of a driver interrupt service routine appears in Section 8.3.

A **timeout handling routine** retries I/O operations and performs other error handling when a device fails to complete a request in a reasonable period of time. Once every second, the system timer checks all devices in the system for device timeout. When it locates a device that has timed out, because it is off line or some error has occurred, the system timer calls the driver's timeout handling routine.

2.4.4 Other Driver Routines

You can also include any of the following routines in a device driver:

Depending upon the reason for the timeout, the timeout handling routine may call a system error-logging routine to allocate and fill an error message buffer with information about the error. In turn, the error-logging routine can call a **register-dumping routine** in the driver that also loads into the buffer the contents of device registers at the time of the error.

The operating system calls a driver's **cancel-I/O routine** when a user process issues a Cancel I/O on Channel ($CANCEL) system service for the device. It may also call the routine when the device's reference count goes to zero, which occurs when all users with assigned channels to the device have deassigned them.

2.5 Driver Context

A driver may have several routines to which the operating system passes control in certain situations. The context in which any one routine receives control from the operating system may differ substantially from that in which another receives control. It is essential that a driver routine not attempt to exceed the limitations of the context in which it executes.

In general, context is characterized by the following factors:

- Actual parameters that are passed to the routine
- The current IPL of the executing processor
- The range of IPLs that the routine can change and the required IPL on return from the routine.
- The currently owned spinlocks of the executing processor
- The data structures available to the routine
- The ability or inability to access process space

A complete description of the context of each driver routine appears in Chapter 18. Here is a general overview:

- All device driver routines execute in kernel mode at an elevated IPL.
- Only driver FDT routines execute within process context and can access process-private address space.
- The majority of driver routines execute in *system* (or *interrupt*) context: that is, in the sequence of execution that follows a processor's grant of an interrupt request at a given IPL. Such code can refer only to system address space. Moreover, it cannot incur exceptions, including page faults, without causing a fatal bugcheck. Code executing in system context is serviced on the kernel stack, and must synchronize its execution with other priority code threads by using IPLs, spinlocks, and resource wait queues, all of which are described in Chapter 3.

Most driver processing of an I/O request (before and after the device acknowledges the servicing of the request by requesting an interrupt from the processor) occurs at a *fork IPL*. This portion of driver code, which includes most of the start-I/O routine, is commonly known as the driver's **fork process**.

There are several instances in the processing of an I/O request when a driver fork process must suspend execution to wait for a resource or a device interrupt. To make the matter of saving and restoring fork process context as efficient as possible, the operating system places a restriction on the context of

a driver fork process, in addition to those that apply to any process in interrupt context. **Fork context** consists of only the following:

- The fork routine parameters (FR3 and FR4)
- The fork routine address (FPC)
- A fork block (usually the unit control block), that can contain additional fork process context

The operating system places the fork block of a suspended fork process in either a processor-specific fork queue or a resource wait queue where it waits to be resumed. When it resumes the fork process, the operating system calls the fork routine with the FR3 value, FR4 value, and a pointer to the fork block as parameters.

2.6 Programmed-I/O and Direct-Memory-Access Transfers

Devices contain registers that initiate, control, and monitor the progress of data transfer, seek operation, or other requests for device activity. When it completes a request, the device posts an interrupt to the processor. The size of the transfer concluded by a device interrupt depends upon the capabilities of the device.

2.6.1 Programmed I/O

Drivers for relatively slow devices, such as printers, terminals, and some disk and tape drives, must transfer data to or from a hardware interface register a byte or a word at a time. These drivers must keep a record of the location of the data buffer in memory, as well as a running count of the amount of data that has been transferred to or from the device. Thus, these devices perform **programmed I/O** (PIO) in that the transfer is largely conducted by the driver. The DE422 ISA ethernet interface is an example of a device that uses programmed I/O. Drivers performing PIO transfers are generally not concerned with the operation of I/O adapters.

2.6.2 Direct-Memory-Access I/O

Devices that perform **direct-memory-access** (DMA) transfers do not require as much driver involvement in the transfer. Once the driver activates the device, the device can transfer a large amount of data without requesting an interrupt after each of the smaller amounts. The responsibilities of a driver for a DMA device involve setting a hardware interface register with the starting address of the buffer containing the data to be transferred, a byte offset into the buffer, and the size of the transfer. By setting the appropriate bit or bits in the hardware interface control and status register (CSR), the driver activates the device. The device then automatically transfers the specified amount of

data to or from the specified address. Any driver that does DMA must map the DMA buffer.

2.7 Buffered and Direct I/O

Another topic, related to the data transfer capabilities of a device, results from the fact that the original buffer, as specified in the user $QIO request, is in process space and is mapped by process page-table entries. Because the driver cannot rely on process context existing at the time the device is ready to service the I/O request, it must have some means of guaranteeing that it can access both the data involved in the transfer and the page-table entries that map the buffer.

The operating system provides the following two techniques that are employed by device drivers:

- **Direct I/O**, the technique used most commonly by drivers of DMA devices, locks the user buffer in memory as well as the page-table entries that map it. The function decision table (FDT) of such a driver calls a system-supplied FDT routine that prepares the user buffer for direct I/O.
- **Buffered I/O** is the strategy whereby the driver FDT dispatches to an FDT routine in the driver that allocates a buffer from nonpaged pool. It is this intermediate buffer that is involved in the transfer. The driver later refers to the buffer using addresses in system space. Driver preprocessing routines copy the data from the user buffer to the system buffer for a write request; system I/O postprocessing (by means of a special kernel-mode AST) delivers data from the system buffer to the user buffer for a read request. Drivers most often use buffered I/O for PIO devices such as line printers and card readers.

The trade-off between buffered I/O and direct I/O is the time required to move the data to or from the user's buffer as against the time required to lock the buffer pages in memory.

2.8 Example of an I/O Request

This section briefly summarizes the processing of a sample I/O request.

1. **A process requests an I/O operation.**

   A user process initiates an I/O request by issuing either a $QIO system service call or an RMS call resulting in a call to the $QIO system service.

   The user process specifies the target device, a read function code, and the address of a buffer into which the data is to be read.

2. **The operating system performs I/O preprocessing.**

The $QIO system service validates the request and locates data structures in the I/O database that describe the device and its driver. The system service also allocates and initializes an I/O request packet to contain a description of the I/O request. The system service then calls a reading routine in the driver.

3. **The driver performs I/O preprocessing.**

   The driver FDT routine verifies that the user buffer resides in virtual memory pages that can be modified by the requesting process, locks the buffer pages in memory, and adds details of the I/O operation to the I/O request packet. The read FDT routine then calls the operating system to send the I/O request packet to the driver.

4. **The system creates a driver's fork process.**

   A system routine creates a fork process in which the device driver can execute. The routine activates the driver's fork process by transferring control to the driver's start-I/O routine.

5. **The driver prepares to transfer data to the I/O adapter.**

   For DMA transfers, the driver's fork process calls system routines that enable the I/O adapter hardware to map I/O bus addresses into physical addresses for the transfer.

6. **The driver activates the device.**

   The fork process prepares the device to begin processing the request by setting bits in device registers.

7. **The driver waits for an interrupt.**

   A system routine saves the context of the driver's fork process and relinquishes the processor until an interrupt occurs.

8. **The device issues an interrupt.**

   When the data transfer is complete, the device issues a hardware interrupt that causes the system to dispatch to the driver's interrupt service routine.

9. **The driver services the interrupt.**

   The driver's interrupt service routine handles the interrupt and reactivates the driver, which reads device registers and possibly data structures to obtain status information about the transfer.

10. **The operating system inserts the driver in a fork queue.**

    The driver requests that it again be suspended, to be reactivated later at a lower software interrupt priority level (IPL).

11. **The fork dispatcher reactivates the driver's fork process.**

When processor priority permits, the system fork dispatcher reactivates the driver as a fork process.

12. **The driver completes the I/O operation.**

    The driver's fork process completes device-dependent processing of the I/O request and returns an I/O status to the operating system.

13. **The operating system completes the I/O operation.**

    The system I/O postprocessing routines copy the I/O status into process address space and return control to the user process.

Only four of these 13 steps describe the driver's I/O preprocessing and fork processing. The system I/O-support routines perform I/O processing common to many I/O requests. Driver writing is further simplified by the use of system routines that handle device-independent functions.

Note that this example simplifies the processing of an I/O operation by ignoring such issues as

- The association of a device with a process, which is to say, device assignment
- Simultaneous I/O requests for one device
- System synchronization issues, such as IPLs and spinlocks
- Driver competition for shared system and I/O adapter resources
- Driver competition for a multiunit controller
- Driver recovery from device errors or power failure

3

Synchronization of I/O Request Processing

Because a device driver executes as kernel-mode code and its execution is triggered by device- and software-initiated interrupts, it can preempt core system tasks and access critical system data. As a result, it must adhere to a set of rules that governs the priority of system activities and controls the flow of system events. These synchronization rules ensure that both the operating system and the device driver access memory in an orderly and consistent fashion.

This chapter discusses synchronization concepts most important to device-driver writers. For more comprehensive information about synchronization concepts and techniques, refer to the *OpenVMS Alpha Internals and Data Structures* book.

3.1 Interrupt Priority Levels

The Alpha architecture defines 32 levels of hardware priority, called interrupt priority levels (IPLs). These IPLs govern the sequence of system events that occur on each processor in an Alpha system. The higher-numbered IPLs (16 to 31) are reserved for hardware interrupts, and the lower-numbered IPLs (1 to 15) are reserved for software interrupts. Most process-based software and all user-mode code runs at IPL 0.

The hardware IPLs (16 to 31) are used for device interrupts (IPLs 20 to 23), interprocessor interrupts in a multiprocessing system, interval timer interrupts, urgent conditions like power failure, and such serious errors as a machine check. For specific hardware IPL information, see your system's hardware documentation.

The software IPLs (1 to 15) are defined by the operating system, as illustrated in Table 3–1.

Table 3–1 System-Defined IPLs

IPL	Symbolic Name	Use
0	—	Execution of most process-based software
1	—	Reserved
2	IPL$_ASTDEL	Servicing of AST-delivery interrupts
3	IPL$_RESCHED	Servicing of scheduler interrupts
4	IPL$_IOPOST	Servicing of I/O-postprocessing interrupts
5	—	Reserved
6	IPL$_QUEUEAST	Fork level processing for queuing ASTs
7	IPL$_TIMERFORK	Entry level for software timer interrupt servicing
8	IPL$_SYNCH	Synchronization of access-to-system databases in a uniprocessor system[1]
11	IPL$_MAILBOX IPL$_POOL	Fork level processing for access to mailboxes Allocation of nonpaged pool
8–11	—	Fork level processing for executing driver code
12	—	Recalculation of quorum; cancellation of mount verification (IPC)
13	—	Reserved
14	—	Entry level for XDelta debugger
15	—	Reserved

[1]IPL$_TIMER, IPL$_SCHED, IPL$_JIB, IPL$_MMG, IPL$_FILSYS, and IPL$_IOLOCK8 are all synonyms for IPL$_SYNCH (see Table 3–3).

Because a higher IPL takes precedence over a lower IPL, a routine executing at one IPL can block interrupts on a processor at that IPL and all lower IPLs. This scheme allows the operating system to assign the higher IPLs to system activities that must be dispatched quickly and with little chance of interruption. In a general sense, each processor services interrupts according to the following priorities:

- Power failure
- Processor errors
- Device interrupts
- Device driver fork processing

- I/O postprocessing
- Process rescheduling
- AST delivery

The mechanism by which synchronized access to data is ensured is twofold. First, the operating system associates a given IPL with the access of one or more data structures or databases. Secondly, the operating system defines an ordered set of semaphores, called **spinlocks**, that extend IPL synchronization throughout a multiprocessing system. A processor must obtain one or more of these spinlocks before executing any code thread that must make use of the resources the spinlock protects. Spinlocks thus allow each processor in a multiprocessing system to share common system data and block events systemwide.

For example, consider a code thread running at IPL 4 that intends to access the memory management database. To do so, it raises IPL to IPL$_MMG. This action gives it the exclusive right to access the database from the local processor, effectively preventing access by other code threads on the same processor. After raising IPL, this code thread requests the memory management (MMG) spinlock. Ownership of the MMG spinlock gives the processor executing this thread the exclusive right to access the database systemwide, and bars access from any other code thread running on any other processor in the Alpha system.

Although discussions in this book treat IPL and spinlock synchronization as conceptually separate tasks for a device driver, system synchronization macros and routines make adjustment of IPL and disposition of spinlocks appear as a single operation.

A full description of spinlocks appears in Section 3.2.

3.1.1 Interrupt Service Routines

The operating system associates certain IPLs with the execution of certain tasks. Moreover, when a processor in a Alpha system grants an interrupt at a given IPL, the grant actually triggers the execution of a specific piece of code, called an interrupt service routine (ISR).

Device drivers themselves contain an interrupt service routine that handles device interrupts at an appropriate device IPL (IPLs 20 to 23). In addition, drivers rely heavily upon the system interrupt service routine known as the **fork dispatcher** that runs at several IPLs, including driver fork IPLs 8 to 11. When the local processor's IPL drops to fork IPL, it is the fork dispatcher that restores the context of the driver fork process and places it into execution.

3.1.2 IPL Use During I/O Processing

The activities essential to the processing of an I/O request occur only at certain IPLs. The operating system performs some of these tasks in system routines and interrupt service routines; drivers perform others. This section describes those IPLs and interrupt service routines that are most involved in processing an I/O request or are of particular interest to device drivers.

3.1.2.1 IPL 2 (IPL$_ASTDEL)

The asynchronous system trap (AST) delivery interrupt service routine (SCH$ASTDEL) is associated with IPL$_ASTDEL.

When an AST is specified for delivery to a process, the AST queuing routine (SCH$QAST) queues the AST to the specified process's process control block (PCB).[1] The mode of the AST, the current mode of the processor, a currently active AST, or a disabled AST recognition determine when the AST is delivered.

The AST delivery interrupt service routine gains control when the processor's IPL drops below IPL$_ASTDEL, and delivers all deliverable ASTs to the currently scheduled process. Any code executing at IPL$_ASTDEL or higher blocks the execution of this interrupt service routine.

To block the delivery of ASTs—specifically the kernel-mode AST that causes process deletion—I/O preprocessing, from the time that the $QIO system service allocates an IRP through the execution of the last FDT routine, occurs at IPLs no lower than IPL$_ASTDEL. Because the system allocation routine records the address of the system memory allocated for the IRP in a local variable, if an AST that deletes the process were to occur, the allocated memory would be lost from the pool.

In addition, some I/O postprocessing occurs in a special kernel-mode AST servicing routine that also executes at IPL$_ASTDEL. The special kernel-mode AST, running in the context of a process whose I/O has been completed, writes status information into an I/O status block, copies buffered input into process space, and deallocates system buffers. The completion of these tasks depends on the availability of process context.

Page faults may be taken by code that executes at IPL$_ASTDEL. However, this is not the case with code executing at higher IPLs. Thus, programs that are sensitive to the contents of pageable data structures run at IPL$_ASTDEL to take page faults. For example, the allocation of paged pool is one such program code thread; paged pool, as a result, is protected by a mutex.

---

[1] Because the system AST queuing and delivery routines access the scheduler database, they synchronize within a multiprocessing environment by obtaining the SCHED spinlock before modifying system data.

3.1.2.2 IPL 4 (IPL$_IOPOST)

The IPL$_IOPOST interrupt service routine (IOC$IOPOST) performs device-independent postprocessing of an I/O request. As appropriate to the I/O request, it adjusts process quota use and deallocates system memory. IOC$IOPOST also queues a special kernel-mode AST to the process's PCB that, once process context is restored, writes status and data into the process's address space.

After it has completed whatever device-dependent postprocessing is required, a driver fork process requests I/O postprocessing by calling a system routine (IOC_STD$REQCOM) that inserts an IRP in the systemwide I/O postprocessing queue and requests a software interrupt at IPL$_IOPOST. When IPL drops below IPL 4, the IPL$_IOPOST interrupt service routine dequeues an IRP from the I/O postprocessing queue, performs all I/O-completion tasks that can occur without reference to the device's unit control block (UCB) and, thus, at an IPL lower than fork IPL.

I/O postprocessing runs at an IPL higher than IPL$_RESCHED so that all pending I/O-completion processing is finished before the scheduler looks for a new process to schedule. The ability of a process to execute can depend on the completion of the postprocessing of an I/O request. Additionally, I/O postprocessing can queue ASTs to certain processes, thus changing their state to computable and resulting in a priority boost. Because all I/O completions are accomplished before rescheduling activities, the scheduler can select from a potentially larger set of computable processes, using more up-to-date information about these processes.

3.1.2.3 IPL 8 (IPL$_SYNCH)

IPL$_SYNCH is the level at which the databases that record and control system functions are synchronized. Individual spinlocks, such as the JIB, SCHED, MMG, and TIMER spinlocks, provide synchronized access to individual databases in a multiprocessing environment.[3] When a system subroutine or a driver needs to modify or read a dynamic portion of a system database, the routine always executes at IPL$_SYNCH, holding an appropriate system spinlock, to ensure that the database does not change because of some interrupt service routine or process action.

[3] IPL$_TIMER, IPL$_SCHED, IPL$_SCS, IPL$_JIB, IPL$_MMG, IPL$_FILSYS, and IPL$_IOLOCK8 are all synonyms for IPL$_SYNCH (see Table 3–3).

3.1.2.4 IPL 6 and IPL 8 to IPL 11 (Fork IPLs)

On each processor in an Alpha system—for IPL 6 and for each of the IPLs from 8 to 11—there exists a queue for fork blocks waiting to be processed. Each fork block contains the context of a suspended fork process. The interrupt service routine that executes at each of these IPLs (EXE$FORKDSPTH) is known as the **fork dispatcher**. The fork dispatcher dequeues a fork block, obtains the appropriate fork lock, restores the context of the fork process, and resumes its execution at the routine address saved in the fork block (at FKB$L_FPC).

All driver routines, except most FDT routines, execute at fork IPL or higher. Usually driver routines should not read or alter UCB fields without taking steps to ensure synchronization. Because such UCB fields can be shared among driver fork processes and system tasks executing on other processors in a multiprocessing system, a processor must first secure the corresponding fork lock to execute at that fork IPL.

A driver places a fork lock index in the `ucb$b_flck` field in its structure initialization routine. (See Section 10.3.) The operating system determines the appropriate fork IPL from the contents of the `spl$b_ipl` field in the fork lock's structure. (See Section 3.2 for a discussion of spinlocks.) Most OpenVMS drivers use the SPL$C_IOLOCK8 spinlock, which uses the IPL 8 fork IPL (IPL$_IOLOCK8). (IPL 6 is also called IPL$_QUEUEAST for historical reasons.)

3.1.2.5 IPL 20 to IPL 23 (Device IPLs)

Alpha peripheral devices request interrupts at IPLs 20 to 23 because device interrupts usually need to preempt most user and system software functions. When a device requests an interrupt at one of these IPLs and the processor is executing at a lower IPL, the processor grants the interrupt, and then transfers control to an interrupt service routine for the device located in its driver. If the processor is executing at a higher or equal IPL, the interrupt remains pending.

The **interrupt dispatcher** routes interrupts from devices to the appropriate device driver's interrupt service routine. A driver specifies the address of its interrupt service routine by using the `dpt_store_isr` macro in its structure re-initialization routine. The interrupt dispatcher's routing mechanism works differently depending upon the Alpha processor and I/O subsystem in use.

Data in a device's registers and in various fields of the UCB that record device status is synchronized on the local processor at device IPL, at which its driver's interrupt service routine executes. This value is stored by the driver in the `ucb$b_dipl` field of the UCB. It is the responsibility of the interrupt service routine to secure the corresponding device lock. This action allows it

to synchronize with other code threads that access the same resources in a multiprocessing system.

The driver's start-I/O routine is one such code thread and must similarly synchronize. The start-I/O routine must secure the appropriate device lock to achieve systemwide synchronization of the device database. The act of acquiring the device lock automatically sets IPL to device IPL.

Because code executing at IPLs 20 to 23 blocks most other hardware interrupts and all software interrupts on the local processor, driver code lowers its IPL as soon as possible.

3.1.2.6 IPL 31 (IPL$_POWER)

The highest IPL, IPL$_POWER (IPL 31), locks out all other interrupts on the local processor. Many system routines and drivers raise IPL to IPL$_POWER to execute code sequences that cannot tolerate interruption. For example, much of system initialization occurs at IPL$_POWER. In a multiprocessing system, these routines often need to acquire additional synchronization, as described in Section 3.2.

When a device driver needs to execute a series of instructions without interruption, the driver raises IPL to IPL$_POWER. The driver should never remain at IPL$_POWER for more than a few instructions. The most common instance of a driver's raising IPL to IPL$_POWER is to determine whether a power failure has occurred on the local processor between the time that the driver writes setup data into device registers and the time that the driver starts the device by writing into the device's control register.

3.1.2.7 IPL 11 (IPL$_MAILBOX)

IPL$_MAILBOX, which is the highest fork IPL, is the fork IPL used by the Mailbox driver.

3.1.3 Modifying IPL in Driver Code

Kernel-mode code can modify the IPL of the local processor by either explicitly setting the processor's IPL to a specific value or by requesting a software interrupt at a specific level. Driver code can change the IPL at which it executes by invoking a system-supplied macro to request a change in IPL. Because the `device_lock`, `fork_lock`, and `sys_lock` macros (and their counterparts) raise (or lower) IPL in a uniprocessing environment, and achieve full synchronization in a multiprocessing system, Digital recommends their use instead of the `setipl`, `dsbint`, and `enbint` macros.

The lock, fork_lock, and device_lock macros ensure that the synchronization needed for either the uniprocessor or multiprocessor environment is obtained before the requested resource is accessed. When executed in a uniprocessor environment, these macros only obtain the proper IPL synchronization. When invoked in a multiprocessing environment, these macros both raise IPL and obtain an appropriate spinlock, thus extending IPL synchronization systemwide.

3.1.3.2 Lowering IPL

Driver code lowers its IPL to synchronize with code threads that access common data or perform common activities at the lower IPL. In a multiprocessing environment, lowering IPL is often associated with the release of a spinlock. In addition, lowering IPL may be necessary in order to obtain a spinlock synchronized at the lower IPL.

One of the most fundamental coding rules under the operating system is that *a code thread cannot explicitly lower IPL below the level at which its execution has been initiated.* In relation to driver processing, this means that a driver fork process cannot explicitly set IPL to be less than its fork IPL, nor can a driver's interrupt service routine explicitly set IPL to be less than device IPL. This is because a processor interrupted a lower IPL code thread in mid-execution to place the current code thread into execution. It is important to the integrity of the data structures protected at this lower IPL that the previous code thread be resumed before other code accesses the same structures. A violation of the IPL rule would undermine the system interrupt dispatching mechanism by not first returning control to the interrupted code thread.

Driver code uses the following methods to lower IPL:

- Issuing a device_unlock, fork_unlock, or sys_unlock macro (paired with an earlier invocation of a device_lock, fork_ock, or sys_lock) or a enbint macro (paired with an earlier invocation of an dsbint macro) to restore IPL to a previously saved value.
- Invoking the iofork (or fork) macro to preserve its context in a fork block, to insert the block in a fork queue, and to request a software interrupt at the driver's fork IPL.
- Returning out of an interrupt service routine to the system interrupt dispatcher. The dispatcher dismisses the interrupt and lowers IPL.

Lowering IPL can cause many pending interrupts on the local processor between the old and new IPLs to become deliverable.

3.2 Spinlocks

In a multiprocessing environment, as in a uniprocessing environment, you can block activities on the local processor by raising IPL. Similarly, certain shared databases must be accessed only at a given IPL. However, in a multiprocessing environment, simply raising IPL on the local processor does not prevent other processors in the system from reading or modifying a shared database. Unless other steps are taken to notify the other processors that the database is "owned," such contention could potentially result in corrupted data and system failures.

A **spinlock** is a semaphore associated with a set of system structures, fields, or registers whose integrity is critical to the performance of a specific operating system task. The scheduler and the memory management subsystem thus have their own spinlocks, as does each fork processing level and each device controller. Because a spinlock can be owned by only one processor in the system at a time, other processors attempting to acquire the same spinlock are prevented from reading from or writing into the database it protects. For more information about the spinlock (SPL) data structure, see Chapter 17.

There are two categories of spinlock:

- The structure of a **static spinlock** is permanently assembled into the system. As a result, its existence and definition are fixed from one system to another. Static spinlocks are accessed as indexes into a vector of longword addresses called the **spinlock vector** and pointed to by SMP$AR_SPNLKVEC. The system spinlocks and fork locks listed in Table 3–3 are static spinlocks.
- A **dynamic spinlock** is a spinlock that is created based on the I/O configuration of a particular system. One such dynamic spinlock is the device lock that is created when particular device is configured. This device lock synchronizes access to the device's registers and certain unit control block (UCB) fields. The operating system creates a dynamic spinlock by allocating space from nonpaged pool, rather than by assembling the lock into the system as it does in the creation of a static spinlock. Section 3.2.2 describes device locks.

Table 3–3 lists, in order of increasing logical rank, the static spinlocks. For each system spinlock or fork lock, the table records its index into the spinlock vector, its synchronization IPL, and a brief description of its function.

The lock, fork_lock, and device_lock macros ensure that the synchronization needed for either the uniprocessor or multiprocessor environment is obtained before the requested resource is accessed. When executed in a uniprocessor environment, these macros only obtain the proper IPL synchronization. When invoked in a multiprocessing environment, these macros both raise IPL and obtain an appropriate spinlock, thus extending IPL synchronization systemwide.

3.1.3.2 Lowering IPL

Driver code lowers its IPL to synchronize with code threads that access common data or perform common activities at the lower IPL. In a multiprocessing environment, lowering IPL is often associated with the release of a spinlock. In addition, lowering IPL may be necessary in order to obtain a spinlock synchronized at the lower IPL.

One of the most fundamental coding rules under the operating system is that *a code thread cannot explicitly lower IPL below the level at which its execution has been initiated.* In relation to driver processing, this means that a driver fork process cannot explicitly set IPL to be less than its fork IPL, nor can a driver's interrupt service routine explicitly set IPL to be less than device IPL. This is because a processor interrupted a lower IPL code thread in mid-execution to place the current code thread into execution. It is important to the integrity of the data structures protected at this lower IPL that the previous code thread be resumed before other code accesses the same structures. A violation of the IPL rule would undermine the system interrupt dispatching mechanism by not first returning control to the interrupted code thread.

Driver code uses the following methods to lower IPL:

- Issuing a device_unlock, fork_unlock, or sys_unlock macro (paired with an earlier invocation of a device_lock, fork_ock, or sys_lock) or a enbint macro (paired with an earlier invocation of an dsbint macro) to restore IPL to a previously saved value.
- Invoking the iofork (or fork) macro to preserve its context in a fork block, to insert the block in a fork queue, and to request a software interrupt at the driver's fork IPL.
- Returning out of an interrupt service routine to the system interrupt dispatcher. The dispatcher dismisses the interrupt and lowers IPL.

Lowering IPL can cause many pending interrupts on the local processor between the old and new IPLs to become deliverable.

3.2 Spinlocks

In a multiprocessing environment, as in a uniprocessing environment, you can block activities on the local processor by raising IPL. Similarly, certain shared databases must be accessed only at a given IPL. However, in a multiprocessing environment, simply raising IPL on the local processor does not prevent other processors in the system from reading or modifying a shared database. Unless other steps are taken to notify the other processors that the database is "owned," such contention could potentially result in corrupted data and system failures.

A **spinlock** is a semaphore associated with a set of system structures, fields, or registers whose integrity is critical to the performance of a specific operating system task. The scheduler and the memory management subsystem thus have their own spinlocks, as does each fork processing level and each device controller. Because a spinlock can be owned by only one processor in the system at a time, other processors attempting to acquire the same spinlock are prevented from reading from or writing into the database it protects. For more information about the spinlock (SPL) data structure, see Chapter 17.

There are two categories of spinlock:

- The structure of a **static spinlock** is permanently assembled into the system. As a result, its existence and definition are fixed from one system to another. Static spinlocks are accessed as indexes into a vector of longword addresses called the **spinlock vector** and pointed to by SMP$AR_SPNLKVEC. The system spinlocks and fork locks listed in Table 3–3 are static spinlocks.
- A **dynamic spinlock** is a spinlock that is created based on the I/O configuration of a particular system. One such dynamic spinlock is the device lock that is created when particular device is configured. This device lock synchronizes access to the device's registers and certain unit control block (UCB) fields. The operating system creates a dynamic spinlock by allocating space from nonpaged pool, rather than by assembling the lock into the system as it does in the creation of a static spinlock. Section 3.2.2 describes device locks.

Table 3–3 lists, in order of increasing logical rank, the static spinlocks. For each system spinlock or fork lock, the table records its index into the spinlock vector, its synchronization IPL, and a brief description of its function.

Table 3–3 Static Spinlocks

Lock Name	Lock Index	Synchronization IPL	Description
QUEUEAST	SPL$C_QUEUEAST	6 (IPL$_QUEUEAST)	Fork lock for executing a fork process at IPL 6
FILSYS	SPL$C_FILSYS	8 (IPL$_FILSYS)[1]	Lock on file system structures
IO_MISC	SPL$C_IO_MISC	8 (IPL$_IO_MISC)	CRAM mailboxes allocation and deallocation
IOLOCK8 SCS	SPL$C_IOLOCK8 SPL$C_SCS	8 (IPL$_IOLOCK8 IPL$_SCS) [1]	Fork lock for executing a fork process at IPL 8
TIMER	SPL$C_TIMER	8 (IPL$_TIMER)[1]	Lock for adding and deleting timer queue entries and searching the timer queue[3]
JIB	SPL$C_JIB	8 (IPL$_JIB)[1]	Lock for manipulating job nonpaged pool quotas as reflected by the fields JIB$L_ BYTCNT and JIB$L_ BYTLM in the job information block
MMG	SPL$C_MMG	8 (IPL$_MMG)[1]	Lock on system memory management, PFN database, swapper, modified page writer, and creation of per-CPU database structures
SCHED	SPL$C_SCHED	8 (IPL$_SCHED)[1]	Lock on process control blocks, scheduler database, and mutex acquisition and release structures
IOLOCK9	SPL$C_IOLOCK9	9 (IPL$_IOLOCK9)	Fork lock for executing a fork process at IPL 9

[1]IPL$_TIMER, IPL$_SCHED, IPL$_SCS, IPL$_JIB, IPL$_MMG, IPL$_FILSYS, and IPL$_IOLOCK8 are all synonyms for IPL$_SYNCH.

[3]The HWCLK spinlock implicitly locks the timer queue element at the head of the timer queue by locking the quadword representing its due time (EXE$GQ_1ST_TIME).

(continued on next page)

Table 3–3 (Cont.) Static Spinlocks

Lock Name	Lock Index	Synchronization IPL	Description
IOLOCK10	SPL$C_IOLOCK10	10 (IPL$_IOLOCK10)	Fork lock for executing a fork process at IPL 10
IOLOCK11	SPL$C_IOLOCK11	11 (IPL$_IOLOCK11)	Fork lock for executing a fork process at IPL 11
MAILBOX	SPL$C_MAILBOX	11 (IPL$_MAILBOX)	Lock for sending messages to mailboxes
POOL	SPL$C_POOL	11 (IPL$_POOL)	Lock on nonpaged pool database
PERFMON	SPL$C_PERFMON	15 (IPL$_PERFMON)	Lock for I/O performance monitoring
INVALIDATE	SPL$C_INVALIDATE	21 IPL$_INVALIDATE	Lock for system space translation buffer (TB) invalidation
HWCLK	SPL$C_HWCLK	22	Lock on interval clock database, including the quadword containing the due time of the first timer queue element and the quadword containing the system time
MEGA	SPL$C_MEGA	31 (IPL$_MEGA)	Lock for serializing access to fork and wait queue
MCHECK EMB	SPL$C_MCHECK SPL$_EMB	31 (IPL$_MCHECK IPL$_EMB)	Lock for synchronizing certain machine error handling and for allocating and releasing error-logging buffers

Drivers rarely need to obtain system spinlocks or fork locks explicitly; the system routines that initiate driver processing and access resources protected by a spinlock generally obtain and release these locks as required. However, a driver must obtain the appropriate device locks whenever it must access data synchronized at device IPL; for instance, in its interrupt service routine.

The operating system provides a set of macros, listed in Table 3–2, that call the system's spinlock acquisition and releasing routines.

Three factors control the successful acquisition of a spinlock: IPL, rank, and ownership.

IPL

The processor must be executing at an IPL equal to or below the spinlock's synchronization IPL (`spl$b_ipl`). In keeping with the rules discussed in Section 3.1.3.2, a processor should not lower the IPL of its thread of execution in the process of acquiring a spinlock. Thus, in acquiring a spinlock, a processor may or may not raise its IPL, depending upon whether it is executing already at the spinlock synchronization IPL. The operating system supplies spinlock acquisition macros (`device_lock`, `fork_lock`, and `sys_lock` that, in calling appropriate system routines, raise IPL automatically in the course of obtaining the requested spinlock. Once it owns the spinlock, the processor can raise its IPL above the IPL at which the spinlock was acquired, but it should not lower it below that level.

Rank

A processor can own multiple spinlocks simultaneously, but must obtain these spinlocks in increasing order of rank. (Table 3–3 lists the spinlocks in order of rank.) In other words, a processor that owns one or more spinlocks should not attempt to acquire a spinlock whose logical rank[4] is less than a spinlock it already holds. It does not need to acquire all spinlocks of intervening rank. This rule is meant to avoid potential deadlocks in the acquisition of system spinlocks and fork locks, and does not pertain to device locks. The processor may release spinlocks in any order, as long as any attempt to reacquire those spinlocks acquires them in ascending order.

Note that the concept of rank is independent of IPL. At any given synchronization IPL, there may be many spinlocks, each of which is ranked according to its position in Table 3–3.

Ownership

The spinlock must not be owned by any other processor. If the spinlock is currently owned by another processor, a requesting processor **spin waits** for the lock to become available. That is, it executes in a loop, waiting for the processor that owns the spinlock to release it.

It is legal for a processor to nest acquisitions of a given spinlock. In other words, if a processor attempts to acquire a spinlock that it currently owns, the acquisition will succeed. The operating system provides a mechanism whereby such a processor can release a single acquisition or all acquisitions of a spinlock.

---

[4] The physical rank of a spinlock is the inverse of its logical rank.

3.2.1 Fork Locks

In its simplest form, a fork lock is a static spinlock that synchronizes the right of a fork process to execute at a specified IPL in a multiprocessing system. Fork locks exist for each of the fork IPLs from IPL 8 to 11. A driver indicates the fork lock under which it processes, and by implication its fork IPL, by filling in `ucb$b_flck` in its initialization routine as described in Section 10.3).

Drivers rarely need to obtain a fork lock explicitly. The operating system places the driver fork process into execution (originally by EXE_STD$INSIOQ and, by implication, by IOC_STD$REQCOM) at fork IPL holding the appropriate fork lock. In addition, the fork dispatcher obtains the fork lock associated with the driver fork process before it restores its context and resumes its execution.

Note that, if a driver fork process is not placed into execution by one of these means, it must itself expressly obtain the fork lock.

As an example, consider a driver fork process activated by a timer wakeup associated with a timer queue element (TQE) previously queued by the driver. The software timer fork routine runs at IPL 8 (IPL$_SYNCH) and obtains certain spinlocks prior to dequeuing the TQE and placing it into execution, but it does *not* obtain the driver's fork lock. Thus, even though the driver's fork IPL may be IPL$_SYNCH, the driver will not be properly synchronized at fork level unless it first obtains the appropriate fork lock.

3.2.2 Device Locks

A device lock represents a lock on an individual adapter or controller. A processor executing a code thread that accesses a device's registers or certain fields in its unit control block (UCB) that reflect its status does so while holding the corresponding device lock.

UCBs are protected by a device lock common to all units on the same adapter or common to the entire system, depending upon the type of device. A device lock is dynamically created by the driver loading service ($LOAD_DRIVER) when it creates a channel request block (CRB). This service stores the address of the device lock in the CRB (`crb$l_dlck`) and later copies it to the unit control block (`ucb$l_dlck`) as a UCB is created for each unit on the controller.

The acquisition of device locks is exempt from the spinlock rank rule. As long as the processor does not violate IPL synchronization, it may successfully obtain an unowned device lock while holding any system spinlock and, likewise, may successfully obtain unowned system spinlocks while holding a device lock. However, a processor can acquire only one device lock at a given IPL.

3.3 Enforcing the Order of Reads and Writes

Many multiprocessing systems have 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 visible 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 Memory Barrier (MB) as appropriate, before and after data references. The MB 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. The Memory Barrier (MB) is accessible to C programmers as a builtin function:

```
#include <builtins.h>
void __MB(void);
```

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 must be issued before the driver initiates the write transaction and the device must issue an MB after receiving the start signal but before starting the DMA read.
- 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 before posting a completion interrupt, and the OpenVMS I/O interrupt dispatcher issues an MB 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 prior to obtaining the lock.

Part II

Creating OpenVMS Alpha Device Drivers

Part II explains how to access device interface registers, allocate map registers, and how to code each part of a driver. It includes the following chapters:

- Chapter 4 discusses device driver register access concepts and routines.
- Chapter 5 explains the driver routines available for managing counted resources.
- Chapter 6 briefly reviews the $QIO system service and describes how to use system-provided and driver-specific FDT routines.
- Chapter 7 describes the general start-I/O code path concepts.
- Chapter 8 explains how to use the simple fork mechanism in a start-I/O code path.
- Chapter 9 explains how to use the kernel process mechanism in a start-I/O code path.
- Chapter 10 describes driver initialization macros and routines.

4

Accessing Device Interface Registers

A hardware **control status register (CSR)** is a memory location through which software interfaces with a hardware component. Every hardware component on an OpenVMS Alpha system, including the CPU and memory, has its own set of interface registers.

Most device drivers need to access controller and device registers to perform their functions. OpenVMS Alpha system architectures may define these accesses in different ways. For example, some architectures use dedicated I/O instructions to read and write the registers. Others use a memory-mapped scheme in which the same load and store instructions used to operate on physical memory are used to perform I/O access. Sometimes, these accesses are distinguished from physical memory access by using a different portion of the machine's physical address space.

The portion of a processor's physical address space through which it accesses CSRs is known as its **I/O space**. OpenVMS Alpha systems provide two ways of accessing CSRs:

- Direct access, or memory mapped access
- Hardware I/O mailbox access, or indirect access

Note

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

4.1 Overview of CSR Accessing Mechanisms

A challenge presented by the Alpha architecture is to create software abstractions that hide the hardware mechanisms for accessing I/O space access from the programmer. These software abstractions contribute to driver portability and promote efficiency. Another difficulty is that the Alpha architecture currently 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 to provide byte and word access granularity for I/O adapter register access:

- **Sparse space addressing**, in which 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**, in which upper order bits in the processor's physical address map to an I/O bus address, while lower order bits are used to implement I/O bus byte enable signals. This requires 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. A driver can deliver commands and write data through a hardware I/O mailbox to the interface registers of a device residing on an I/O bus.

The OpenVMS Alpha operating system supports the following basic model for accessing I/O device registers on any system:

- Map the device into the processor address space using CSR mapping routines.
- Access the device using the platform independent access routines IOC$READ_IO and IOC$WRITE_IO or the Controller Register Access Mechanism (CRAM) data structure and associated routines.

The remaining sections in this chapter provide more information about the mapping and access routines available for accessing device registers.

Note

Register mapping is not required on XMI devices on DEC 7000/10000 systems, and the IOC$READ_IO and IOC$WRITE_IO routines are not supported. If you are porting an OpenVMS VAX XMI device driver to an OpenVMS Alpha system, you must use the CRAM data structures and routines.

4.2 Mapping I/O Device Registers

Before you access device registers on OpenVMS Alpha systems, you must map the registers into the processor's virtual address space using the CSR mapping routines described in this section.

Once your device is mapped, you can access it using the IOC$READ_IO and IOC$WRITE_IO routines or the CRAM data structure and associated routines. For more information about using the IOC$READ_IO and IOC$WRITE_IO routines, see Section 4.3. For more information about using the CRAM data structure and associated routines, see Section 4.4.

4.2.1 Using the IOC$MAP_IO Routine

The platform independent I/O bus mapping routine IOC$MAP_IO maps I/O bus physical address space into an address region accessible by the processor. The caller of this routine can express the mapping request in terms of the bus address space without regard to address swizzling, dense space, or sparse space. The IOC$UNMAP_IO routine is provided to unmap a previously mapped space.

In the following example, the IOC$READ_PCI_CONFIG routine locates the memory address to map. The IOC$MAP_IO then maps that address into swizzle space, enabling byte and word accesses. IOC$MAP_IO yields an I/O handle, which can then be used in calls to IOC$READ_IO and IOC$WRITE_IO.

```
status = ioc$read_pci_config (adp,                  /* ADP                          */
         crb->crb$l_node,                           /* Slot number                  */
         offsetof( PCI, pci$l_base_address_1 ),/* Offset into config space */
         sizeof( uint32 ),                          /* Size to read                 */
         &mem_base );                               /* Base address in memory       */

if bad(status) return status;

mem_base &= PCI$M_BASE_ADDRESS_BITS_31_4;      /* constant = 0xFFFFFFF0; AND out
                                                  the bottom byte!             */

status = ioc$map_io(adp,                            /* ADP                          */
         crb->crb$l_node,                           /* Slot                         */
         &mem_base,                                 /* Base address in memory space */
         sizeof( ISP_REGISTERS ),                   /* Size to map                  */
         IOC$K_BUS_MEM_BYTE_GRAN,                   /* Map to swizzle space         */
         &iohandle );                               /* I/O handle                   */

if bad(status) return status;
```

For more details about the IOC$MAP_IO routine interface, see Chapter 19.

4.2.2 Using the CSR Mapping Routine

Drivers must call the IOC$MAP_IO routine with specific spinlock restrictions; in particular, the driver cannot be holding any spinlocks of higher rank than the MMG spinlock and must be executing at IPL 8 or less. For more information about IPLs and spinlocks, see Chapter 3.

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. A special routine, known as the CSR_MAPPING routine, is called before the controller or unit initialization routine. The CSR_MAPPING routine establishes an environment where IOC$MAP_IO can be safely called.

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

There are two important elements of the CSR mapping routine:

1. A device driver may request preallocated space for any number of I/O handles (the output of IOC$MAP_IO).
2. A device driver may name a routine that will be called in an environment suitable for calls to IOC$MAP_IO.

Drivers can specify the number of I/O handles they need to store by using the `ini_dpt_iohandles` macro in the DRIVER$INIT_TABLES routine. 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. MCJ$Q_ENTRIES is the base address in the MCJ of an array of quadword I/O handle slots. The number of slots in the array is exactly the number specified by the drivers dpt$iohandles value.

4.3 Using Platform Independent I/O Access Routines

The IOC$READ_IO and IOC$WRITE_IO access routines provide an easy-to-use, platform independent way to read and write I/O space.

The IOC$READ_IO and IOC$WRITE_IO routines require that the I/O space to be accessed has been previously mapped by a call to the IOC$MAP_IO routine. Both IOC$READ_IO and IOC$WRITE_IO require the IO handle returned from the IOC$MAP_IO routine.

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

The following examples of the IOC$READ_IO and IOC$WRITE_IO routines show the read and write of a byte-wide register. The io_handle was saved from a previous call to the IOC$MAP_IO routine. After the read the register_ contents variable will contain the byte in the appropriate byte lane. In this case, in byte 2, which is the lane into which data is shifted before the call to write the data.

```
status = ioc$read_io (ucb_ptr->ucb$r_ucb.ucb$ps_adp,/* ADP          */
         &io_handle,                        /* I/O handle          */
         REG_OFFSET,                        /* Register Offset     */
         1,                                 /* Number of bytes     */
         &register_contents);               /* Contents of register */

if bad (status) return status;

write_register = write_data << 2;          /* Shift the write data  */

status = ioc$write_io ( ucb_ptr->ucb$r_ucb.ucb$ps_adp,  /* ADP     */
         &io_handle,                        /* I/O handle          */
         REG_OFFSET,                        /* Register Offset     */
         1,                                 /* Number of bytes     */
         &write_register );                 /* Register with write data */

if bad (status) return status;
```

For more details about the IOC$READ_IO and IOC$WRITE_IO routine interfaces, see Chapter 19.

4.4 Using the Controller Register Access Mechanisms (CRAMs)

A controller register access mechanism (CRAM) is a data structure used for reading and writing I/O space. Table 4–1 lists the routines and macros that create and operate the CRAM data structure. For more information about each system routine, see Chapter 19. The subsequent sections in this chapter describe driver mailbox operations in more detail.

Table 4–1 OpenVMS System Routines That Manage I/O Mailbox Operations

Routine	Description
IOC$ALLOCATE_CRAM	Allocates and initializes a CRAM
IOC$CRAM_CMD	Generates values for the command, mask, and remote I/O interconnect address (RBADR) fields of a CRAM
IOC$CRAM_IO	Issues the I/O space transaction defined by the CRAM
IOC$DEALLOCATE_CRAM	Deallocates a CRAM

4.4.1 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 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.

A driver should not reuse a CRAM until it has checked the return status from IOC$CRAM_IO to be certain the register access is complete.

4.4.1.1 Preallocating CRAMs to a Unit or Controller

Device drivers often preallocate CRAMs to perform I/O operations on unit or controller registers. A 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 returned CRAM in the CRAM list. Alternatively, CRAMs can be automatically preloaded by driver loading.

The driver loading procedure examines two fields in the DPT, DPT$W_IDB_CRAMS and DPT$W_UCB_CRAMS to determine how many CRAMs to preallocate. Although the default value of both fields is zero, you can initialize them by specifying the `ini_dpt_idb_crams` and `ini_dpt_ucb_crams` macros in the DRIVER$INIT_TABLES routine. 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 list head for unit CRAMs, and IDB$PS_CRAM as the list head for controller CRAMs.

4.4.1.2 Calling IOC$ALLOCATE_CRAM to Obtain a CRAM

To allocate a single CRAM, a driver calls 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

Usually, a driver can use `ini_dpt_idb_crams` and `ini_dpt_ucb_crams` macros in the DRIVER$INIT_TABLES routine to allocate CRAMs and associate them with a UCB or IDB; drivers that need to associate CRAMs with other structures may 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.

4.4.2 Constructing a Mailbox Command Within a CRAM

Once a driver has allocated CRAMs for its operations on device registers, it initializes each CRAM for access to a device interface register.

To initialize a CRAM, a driver initializes a CRAM by calling IOC$CRAM_CMD, specifying the **cmd_index**, **byte_offset**, **adp_ptr**, **cram_ptr**, and **iohandle** arguments. These arguments supply values for the command, mask, and I/O bus address fields of the CRAM that are specific to the bus being accessed.

Use the **cmd_index** argument to indicate the size and type of the register operation the mailbox describes. Although the cramdef.h system header file defines the command indices listed in Table 4–2, 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 the SS$_BADPARAM status.

Table 4–2 Mailbox Command Indices Defined by cramdef.h

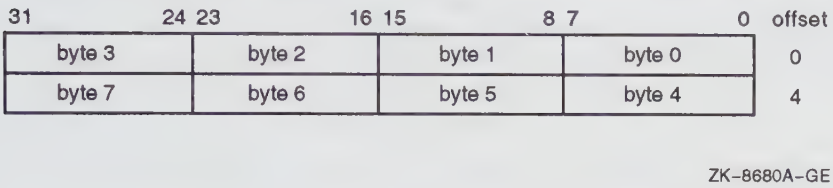
Command Index	Description
CRAMCMD$K_RDQUAD32	Quadword read in 32-bit space
CRAMCMD$K_RDLONG32	Longword read in 32-bit space
CRAMCMD$K_RDWORD32	Word read in 32-bit space
CRAMCMD$K_RDBYTE32	Byte read in 32-bit space
CRAMCMD$K_WTQUAD32	Quadword write in 32-bit space
CRAMCMD$K_WTLONG32	Longword write in 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.

4.4.2.1 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:



ZK-8680A-GE

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).

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.

4.4.3 Initiating a Mailbox Transaction

A driver initiates access to a device register by using IOC$CRAM_IO.

5
Allocating Map Registers

For Alpha I/O subsystems that provide map registers, OpenVMS Alpha provides a set of a routines to manage the allocation of any resource that shares 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.

Two OpenVMS Alpha data structures record information about counted resources.

- 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 the `adp$l_crab` field.

  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.

An OpenVMS Alpha device driver performs the following tasks when setting up and completing a direct memory access (DMA) transfer:

1. Calls the IOC$ALLOC_CRCTX routine to obtain a CRCTX that describes a request for map registers.
2. Loads the request count into the `crctx$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 the IOC$DEALLOC_CNT_RES routine to free the registers for use by other requesters.
7. Calls the IOC$DEALLOC_CRCTX routine to deallocate the CRCTX.

The following sections describe these steps.

5.1 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$ALLOC_CRCTX requires as input the address of the CRAB that describes the counted resource. For adapters that provide a counted resource, such as a set of map registers, the `adp$l_crab` field 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:

```
status = ioc$alloc_crctx (adp->adp$l_crab, ucb->ucb$l_crctx)
```

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.

5.2 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$l_item_cnt	Number of items to be allocated. When requesting map registers, the value in this field should include two extra map registers to be allocated and loaded as a guard page to prevent runaway transfers. There may be additional bus-specific requirements, as described in the bus-specific chapters in this book.
crctx$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$ALLOC_CNT_RES always returns to its caller immediately, whether the allocation request is granted immediately, is stalled, or is unsuccessful. Whether the request is granted immediately or stalled and eventually granted, IOC$ALLOC_CNT_RES returns the number of the first item granted 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$ALLOC_CNT_RES saves up to 3 quadwords of caller-supplied context in the CRCTX. IOC$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$ALLOC_CNT_RES is now able to grant the requested number of resource items, it calls the callback routine (CRCTX$L_CALLBACK), passing it the following values:

Location	Contents
STATUS	SS$_NORMAL
ADP$L_CRAB	Address of CRAB
UCB$L_CRCTX	Address of CRCTX that was specified by IOC$ALLOC_CNT_RES
CRCTX->CRCTX$Q_CONTEXT1	Contents of CONTEXT1 argument at the time of the original allocation request (CRCTX$Q_CONTEXT1)
CRCTX->CRCTX$Q_CONTEXT2	Contents of CONTEXT2 argument at the time of the original allocation request (CRCTX$Q_CONTEXT2)
CRCTX->CRCTX$Q_CONTEXT3	Contents of CONTEXT3 at the time of the original allocation request (CRCTX$Q_CONTEXT3)

The callback routine checks STATUS 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 following example illustrates the call of IOC$ALLOC_CNT_RES in SYS$PKQDRIVER.EXE. (This device driver is available in the OpenVMS Alpha source listings kit.)

```
/*  If the data transfer falls outside the direct DMA window
 *  for the platform, use the map registers.
 *
 *  The CRCTX fields must be initialized before the map registers
 *  can be allocated.
 *
 *  The number of resources to allocate is determined by manipulating
 *  fields within the SCDRP and the SPDT.
 */
if( use_map_registers == TRUE ) {
    crctx = ( CRCTX * )&scdrp->scdrp$r_crctx_base;
    crctx->crctx$w_size = CRCTX$K_LENGTH;
    crctx->crctx$b_type = DYN$C_MISC;
    crctx->crctx$b_subtype = DYN$C_CRCTX;
    crctx->crctx$l_crab = spdt->basespdt.spdt$ps_crab;
    crctx->crctx$b_flck = spdt->basespdt.spdt$is_flck;
    crctx->crctx$l_flags = priority;
    crctx->crctx$l_aux_context = scdrp->scdrp$ps_kpb;
    crctx->crctx$l_item_num = 0;
    crctx->crctx$l_up_bound = 0;
    crctx->crctx$l_low_bound = 0;
    crctx->crctx$l_callback = pkq_buffer_map_restart;
    crctx->crctx$l_saved_callback = 0;
    crctx->crctx$l_item_cnt =
        ((((  scdrp->scdrp$l_boff & spdt->basespdt.spdt$is_crctx_bwp_mask ) +
           scdrp->scdrp$l_bcnt + spdt->basespdt.spdt$is_crctx_bwp_mask ) >>
          spdt->basespdt.spdt$is_crctx_shift ) +
         spdt->basespdt.spdt$is_extmapreg );

    /* Allocate the mapping resources.  If the allocation fails and the
     * request was high priority, convert it to normal priority and try
     * the allocation again.  If allocation fails again, stall the thread
     * until the map restart routine notifies us that the resources have
     * been allocated for us.
     */
    if( ERROR( status = ioc$alloc_cnt_res( spdt->basespdt.spdt$ps_crab, crctx, 0, 0, 0 )))
        if( crctx->crctx$l_flags & CRCTX$M_HIGH_PRIO ) {
            crctx->crctx$l_flags &= ~CRCTX$M_HIGH_PRIO;
            status = ioc$alloc_cnt_res( spdt->basespdt.spdt$ps_crab, crctx, 0, 0, 0 );
        }
        if( ERROR( status )) {
            kpb = scdrp->scdrp$ps_kpb;
            kpb->kpb$ps_sch_stall_rtn = ioc$return;
            kpb->kpb$ps_sch_restrt_rtn = 0;
            exe$kp_stall_general( kpb );
        }
    }
```

You can 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$ALLOC_CNT_RES never stalls a high-priority counted resource request or places its CRCTX in a resource-wait queue. Rather, it attempts to allocate the requested number of resource items immediately. If unsuccessful, IOC$ALLOC_CNT_RES returns SS$_INSFMAPREG status to its caller.

5.3 Loading Map Registers

A driver calls IOC$LOAD_MAP to load a set of adapter-specific map registers. The driver must have previously allocated the map registers (including two extra to serve as guard pages) in calls to IOC$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$LOAD_MAP returns to a specified location a port-specific address of a DMA buffer. The following example illustrates a call to IOC$LOAD_MAP:

```
/* The mapping resources are in hand; load the map.  */

   scdrp->scdrp$l_port_boff = scdrp->scdrp$l_boff;
   scdrp->scdrp$l_port_svapte = scdrp->scdrp$l_svapte;
   scdrp->scdrp$l_sva_spte = 0;
   status = ioc$load_map( spdt->basespdt.spdt$l_adp,
                          crctx,
                          scdrp->scdrp$l_svapte,
                          scdrp->scdrp$l_boff,
                          &scdrp->scdrp$l_sva_dma );
```

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.

5.4 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, if there are any stalled requests, IOC$DEALLOC_CNT_RES queues a fork thread that will attempt to allocate the resource to the stalled requests.

The following example illustrates a call to IOC$DEALLOC_CNT_RES:

```
crctx = ( CRCTX * )&scdrp->scdrp$r_crctx_base;
if( SUCCESS( status = ioc$dealloc_cnt_res( spdt->basespdt.spdt$ps_crab, crctx)))
{
          scdrp->scdrp$is_item_cnt = 0;
          scdrp->scdrp$is_item_num = 0;
          scdrp->scdrp$l_sva_dma = 0;
          scdrp->scdrp$l_sva_spte = 0;
    }
```

5.5 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:

```
STATUS = IOC$DEALLOC_CRCTX (UCB -> UCB$L_CRCTX); Deallocate the CRCTX
```

6

Writing FDT Routines

A driver performs device-specific I/O function preprocessing to validate the device dependent parameters specified in the original call to the $QIO system service, to complete certain types of function processing such as set mode and sense mode operations, and to prepare to service functions involving a device transaction. A device driver can include custom FDT routines or use some of the general-purpose routines that are part of the executive.

This chapter briefly reviews the $QIO system service and describes how to use system-provided and driver-specific FDT routines.

6.1 $QIO System Service

The $QIO system service performs device-independent preprocessing and, via FDT routines, device-dependent preprocessing. It then queues an I/O request to the driver for the device associated with a channel. Any additional processing is performed by the device driver's start I/O routine.

The function prototype for the $QIO system service is as follows:

```
int sys$qio (unsigned int efn, unsigned short int chan, unsigned int func,
        struct _iosb *iosb, void (*astadr)(__unknown_params),
        __int64 astprm, void *p1, __int64 p2, __int64 p3, __int64 p4,
        __int64 p5, __int64 p6);
```

- The efn parameter is the number of the event flag to be associated with the I/O request. Since this parameter is passed by value, omitting it is the same as specifying event flag 0.
- The chan parameter is the identifier of the I/O channel. This is the same as the chan parameter returned by the $ASSIGN system service.
- The func parameter identifies what operation is to be performed by the device driver. It is divided into two portions, the function code proper and function modifiers. In this chapter, the term **function code** means just the function code proper; the term **func** means the entire argument.

- The iosb parameter is the address of the IOSB, a quadword to receive final status of the I/O operation.
- The astadr parameter is the address of an AST procedure to be executed in the mode of the requestor when the I/O operation completes.
- The astprm parameter is the parameter to be passed to the AST procedure.
- There are six optional device- and function-specific parameters, p1 through p6, which are validated and processed by FDT routines.

The chan and func arguments must be specified. All others are optional and, if not specified, default to a value of zero.

6.2 Context of Driver FDT Processing

The $QIO system service executes in the context of the process that issues the I/O request, in kernel mode and at IPL$_ASTDEL. Process context allows the $QIO system service and driver FDT routines to access process address space. Because the $QIO system service expects FDT routines to preserve this context, an FDT routine observes the following conventions:

- An FDT routine must not call system services or OpenVMS RMS routines.
- It must not lower IPL below IPL$_ASTDEL. If an FDT routine raises IPL, it must obtain any appropriate spinlock, and it must lower IPL to IPL$_ASTDEL before exiting, releasing any acquired spinlock.
- It should not access device registers because the device might be active.
- It should exercise caution when modifying the UCB. Routines usually access the UCB while holding the associated fork lock at driver fork IPL to synchronize modifications, and FDT routines do not execute with such synchronization. Drivers containing FDT routines that access device registers or carelessly modify the UCB risk unpredictable operation or a system failure.

6.3 FDT Routine Overview

The primary purpose of FDT routines is to validate and process the device-dependent $QIO parameters p1 to p6. Regardless of the location of FDT routines, they are logically device-dependent extensions of the $QIO system service. Driver I/O function preprocessing on OpenVMS Alpha systems typically requires the cooperative efforts of upper-level FDT action routines, FDT support routines, and FDT completion routines. This section briefly describes FDT upper-level, support, and completion routines. The section that follow contain more details about how drivers use these routines.

An **upper-level FDT action routine** is listed in a driver's function decision table (FDT) when a driver uses the `ini_fdt_act macro` in its DRIVER$INIT_TABLES routine. (see Chapter 10 for more information). 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.

The $QIO system service uses the FDT to determine which upper-level FDT action routine to call to initiate driver preprocessing of a specific I/O request. Often a driver can use one of the system-provided upper-level FDT action routines, as described in Section 6.4.1. This practice encourages the use of well debugged routines and minimizes driver size. However, if the I/O function requires preprocessing that is unique to the device the driver controls, the driver includes an upper-level FDT action routine that services the function in a device-dependent manner.

An upper-level FDT action routine sometimes calls an intermediate FDT routine, known as an **FDT support routine**. An FDT support routine performs some discrete action on behalf of an upper-level action routine, such as determining whether a user buffer is accessible, locking a user buffer in memory, and reformatting data into buffers in the system address space. Often, an FDT support routine calls an FDT completion routine.

There is no uniform interface for calling FDT support routines. These routines are listed and described in the Chapter 19. The $QIO system service never calls an FDT support routine directly to process a given I/O function.

To conclude the preprocessing of an I/O function, a driver FDT routine (either the upper-level FDT action routine or an FDT support routine) calls a system-provided **FDT completion routine**. An FDT completion routine places the status return to the $QIO system service in the FDT context (FDT_CONTEXT) structure and returns SS$_FDT_COMPL status to its caller. Eventually, the driver's upper-level FDT action routine exits FDT processing by returning control to the $QIO system service.

Note that FDT support routines and FDT completion routines return status to their callers. Each FDT routine that participates in the processing of an I/O function should examine the status value returned to it by any routine it calls and should reflect this status to the routine that called it.

6.4 Upper-Level FDT Action Routines

An OpenVMS Alpha device driver provides a single upper-level FDT action routine for each I/O function code it processes. The $QIO system service uses the low-order six bits of the I/O function code as an index into the FDT upper-level action routine vector. As described in Chapter 10, any vector slots corresponding to a driver-supported function contain the procedure value of either a system-provided or driver-provided upper-level FDT action routine. Those that correspond to unsupported functions contain the procedure value of the system upper-level FDT action routine EXE$ILLIOFUNC.

The $QIO system service transfers control to an upper-level FDT action routine using the following interface:

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

The parameters include the addresses of:

- The I/O request packet (IRP) for the current I/O request
- The process control block (PCB) of the current process
- The unit control block (UCB) of the device assigned to the process-I/O channel specified as a parameter to the $QIO request
- The channel control block (CCB) that describes the process-I/O channel

An upper-level FDT action routine must return SS$_FDT_COMPL status to its caller, the FDT dispatching code in the $QIO system service.

An OpenVMS Alpha driver obtains the contents of the function-dependent parameter from `irp$l_qio_p`*n*, where *n* is a parameter number from 1 through 6.

Before exiting, the upper-level FDT action routine takes steps to complete FDT processing. Typically, these steps include:

- Calling an FDT completion routine, which takes steps to complete the processing of an I/O request or to deliver the I/O request to the driver. An FDT completion routine typically provides the final $QIO completion status in the FDT_CONTEXT structure and returns SS$_FDT_COMPL warning status to its caller. SS$_FDT_COMPL status is a warning that FDT processing has been completed and that the IRP is no longer accessible to FDT processing. (For instance, the IRP may have been deallocated or queued to the driver's start-I/O routine, which accesses the IRP at fork IPL.)
- Returning SS$_FDT_COMPL status to its caller, FDT dispatching code in the $QIO system service.

Section 6.4.2 describes the FDT completion routines provided by OpenVMS.

6.4.1 System-Provided Upper-Level FDT Action Routines

The system-provided upper-level FDT routines perform I/O request validation that is common to many devices. Whenever possible, drivers should take advantage of these routines. All of the system FDT routines listed in Table 6–1 transfer control to EXE_STD$QIODRVPKT, EXE_STD$FINISHIO, or EXE_STD$ABORTIO to complete the I/O request. These FDT completion routines, as described in Section 6.4.2, place final $QIO completion status in the FDT_CONTEXT structure, and return SS$_FDT_COMPL status to the upper-level FDT action routine. All upper-level FDT action routines return to the FDT dispatching code in the $QIO system service.

For additional information about system-provided FDT routines, see the specific routine descriptions in the Chapter 19.

Table 6–1 System-Provided Upper-Level FDT Action Routines

FDT Routine	Function	Completion Routine Used
ACP_STD$ACCESS	Processes access and create ACP functions	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
ACP_STD$ACCESSNET	Processes a connects to network function	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
ACP_STD$DEACCESS	Processes a deaccess ACP function	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
ACP_STD$MODIFY	Processes delete and modify ACP functions	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
ACP_STD$MOUNT	Processes a mount ACP function	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
ACP_STD$READBLK	Processes a read block ACP function	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO
ACP_STD$WRITEBLK	Processes a write block ACP function	Calls EXE_STD$ABORTIO, EXE_STD$QIODRVPKT, or EXE_STD$FINISHIO

(continued on next page)

Table 6–1 (Cont.) System-Provided Upper-Level FDT Action Routines

FDT Routine	Function	Completion Routine Used
EXE$ILLIOFUNC	Processes I/O functions not supported by the driver	Calls EXE_STD$ABORTIO
EXE_STD$LCLDSKVALID	Processes an IO$_PACKACK, IO$_AVAILABLE, or IO$_UNLOAD function on a local disk	EXE_STD$FINISHIO or EXE_STD$QIODRVPKT
EXE_STD$MODIFY	Processes a logical-read/write or physical-read/write function for a read and write direct I/O operation to a user-specified buffer	Calls EXE_STD$ABORTIO if an error occurs (for instance, if the user I/O buffers cannot be accessed or cannot be locked in memory); otherwise, calls EXE_STD$QIODRVPKT
EXE_STD$ONEPARM	Processes a nontransfer I/O function code that has one parameter associated with it	Calls EXE_STD$QIODRVPKT
EXE_STD$READ	Processes a logical-read or physical-read function for a direct I/O operation	Calls EXE_STD$ABORTIO if an error occurs (for instance, if the user I/O buffers cannot be accessed or locked in memory); otherwise, calls EXE_STD$QIODRVPKT
EXE_STD$SENSEMODE	Processes the sense-device-mode and sense-device-characteristics functions by reading fields of the UCB	Calls EXE_STD$FINISHIO
EXE_STD$SETCHAR[1]	Processes the set-device-mode and set-device-characteristics functions	Calls EXE_STD$FINISHIO
EXE_STD$SETMODE[1]	Processes the set-device-mode and set-device-characteristics functions by creating a driver fork process	Calls EXE_STD$ABORTIO if an error occurs; otherwise, calls EXE_STD$QIODRVPKT

[1] If setting device characteristics requires no device activity or requires no synchronization with fork processing, the driver's FDT entry can specify EXE_STD$SETCHAR; otherwise, it must specify EXE_STD$SETMODE.

(continued on next page)

Table 6–1 (Cont.) System-Provided Upper-Level FDT Action Routines

FDT Routine	Function	Completion Routine Used
EXE_STD$WRITE	Processes a logical-write or physical-write function for a direct I/O operation	Calls EXE_STD$ABORTIO if an error occurs (for instance, if the user I/O buffers cannot be accessed or locked in memory); otherwise, calls EXE_STD$QIODRVPKT
EXE_STD$ZEROPARM	Processes a nontransfer I/O function code that has no associated parameters	Calls EXE_STD$QIODRVPKT

An FDT routine selects an I/O completion path based on the following factors:

- Whether it needs to call another FDT routine to perform additional function-specific processing.
- Whether an error is found in the I/O request.
- Whether the operation is complete.
- Whether the I/O operation requires and is ready for device activity.

Any specific I/O function can be processed by only one upper-level FDT action routine. Although an upper-level routine can call any number of subsequent routines, it must eventually complete I/O processing by returning the SS$_FDT_COMPL status (and final $QIO completion status in the FDT_CONTEXT structure) to its caller, FDT dispatching code in the $QIO system service.

The system-provided upper-level FDT routines, as discussed in Table 6–1, all call an FDT completion routine that queues an IRP, completes an I/O request, or aborts an I/O request. These FDT completion routines insert the final $QIO system service status in the FDT_CONTEXT structure and return SS$_FDT_COMPL warning status to the upper-level FDT action routine. The upper-level FDT action routine returns these status values to its caller, FDT dispatching code in the $QIO system service.

6.4.2 FDT Exit Paths

An upper-level FDT action routine completes an I/O function by invoking one of the completion macros listed in Section 6.4.3. When an FDT routine completes an I/O request, use an FDT completion macro to invoke the FDT completion routines. For example:

```
status = call_qiodrvpkt (irp, (UCB *)ucb);
status = call_finishioc (irp, (UCB *)ucb, SS$_NORMAL);
status = call_abortio (irp, pcb, (UCB *)ucb, status);
```

Once a FDT completion routine is invoked, the IRP must not be accessed by the caller. FDT completion routines and macros return the SS$_FDT_COMPL warning status, which must be returned by all upper-level FDT routines. For example:

```
return call_finishioc (irp, (UCB *)ucb, SS$_NORMAL);
```

6.4.3 FDT Completion Macros and Associated Routines

The FDT completion macros and associated routines are as follows:

call_abortio
Calls EXE_STD$ABORTIO to abort an I/O request.

An FDT routine that discovers a device-independent error should always use this method of exiting. Inability to gain access to a data buffer or an error in the specification of the I/O request are examples of device-independent errors.

call_altquepkt
Calls EXE_STD$ALTQUEPKT to call an alternate start-I/O routine in the driver (specified in the driver dispatch table at offset DDT$PS_ALTSTART_2) that synchronizes requests for activity on a device unit and initiates the processing of I/O requests.

The FDT routine uses this exit method when it has successfully completed all driver preprocessing and the request requires device activity. However, in contrast to EXE_STD$QIODRVPKT, EXE_STD$ALTQUEPKT bypasses the device unit's pending-I/O queue and the device busy flag; thus, the driver is activated regardless of whether the device unit is busy. A driver that can handle two or more I/O requests simultaneously uses this exit method.

Be aware that programming a device driver to process simultaneous I/O requests requires detailed knowledge of the internal design of the operating system. A driver that uses the `call_altquepkt` macro must not only maintain its internal queues but must also synchronize those queues with the unit's pending-I/O queue, which the operating system maintains. In addition, if a driver processes more than one IRP at the same time, it must use separate fork blocks. Such a driver completes the processing of I/O requests by using the `call_post` macro calling the COM_STD$POST routine. This routine places each IRP in the systemwide I/O postprocessing queue and returns control to the driver. The driver can then fetch another IRP from an internal queue. For more information about the COM_STD$POST routine, see Chapter 19.

When the alternate start-I/O routine finishes, it returns control to EXE_STD$ALTQUEPKT. EXE_STD$ALTQUEPKT then returns to the FDT routine that called it. The FDT routine performs any necessary postprocessing, returning the SS$_FDT_COMPL status to FDT dispatching code in the $QIO system service.

call_finishio
Moves the contents of R0 and R1 to `irp$l_iost1` and `irp$l_iost2`, respectively and calls EXE_STD$FINISHIO to insert the IRP in the I/O postprocessing queue. EXE_STD$FINISHIO returns SS$_FDT_COMPL status to the $QIO system service and SS$_NORMAL status (in the FDT_CONTEXT structure) to the caller of $QIO.

An FDT routine that discovers a device-dependent error should always return status using CALL_FINISHIO or CALL_FINISHIOC.

call_finishioc
Calls the EXE_STD$FINISHIO routine to perform the same operations as the `call_finishio` macro, except `call_finishioc` clears the second longword of the final I/O status.

call_finishio_noiost
Calls EXE_STD$FINISHIO to perform the same operations as the `call_finishio` macro, except `call_finishio_noiost` does not fill in the I/O status fields of the IRP.

call_iorsnwait
Calls EXE_STD$IORSNWAIT. Reserved to Digital.

call_qioacppkt
Calls EXE_STD$QIOACPPKT. Reserved to Digital.

call_qiodrvpkt
Calls EXE_STD$QIODRVPKT to transfer control to a system routine (EXE_STD$INSIOQ), that either delivers an IRP immediately to a driver's start-I/O routine or places the IRP in a pending-I/O queue waiting for driver servicing. The FDT routine uses this FDT completion routine if all preprocessing is complete, if no errors are found in the specification of an I/O request, and if device activity, synchronized access to the device's UCB, or synchronized access to device registers is required to complete the I/O request. Common examples of such a request are read and write functions.

EXE_STD$INSIOQ transfers control to the device driver's start-I/O routine only if the device unit is currently idle. If the device unit is busy, EXE_STD$INSIOQ inserts the IRP in a priority-ordered queue of IRPs waiting for the unit.

Once an FDT routine transfers control to EXE_STD$QIODRVPKT, no driver code that further processes the I/O request can refer to process virtual address space. When a device driver's start-I/O routine gains control, the process that queued the I/O request might no longer be the mapped process. Therefore, the driver must assume that all information regarding the I/O request is in the UCB or the IRP and that all buffer addresses in the UCB are either system addresses or page-frame numbers that can be interpreted in any process context.

For direct I/O operations, FDT routines also must have locked all user buffer pages in physical memory because paging cannot occur at driver fork level or higher interrupt priority levels. The process virtual address space is not guaranteed to be mapped again until the operating system delivers a special kernel-mode asynchronous system trap (AST) to the requesting process as part of I/O postprocessing.

6.5 FDT Routines for System Direct I/O

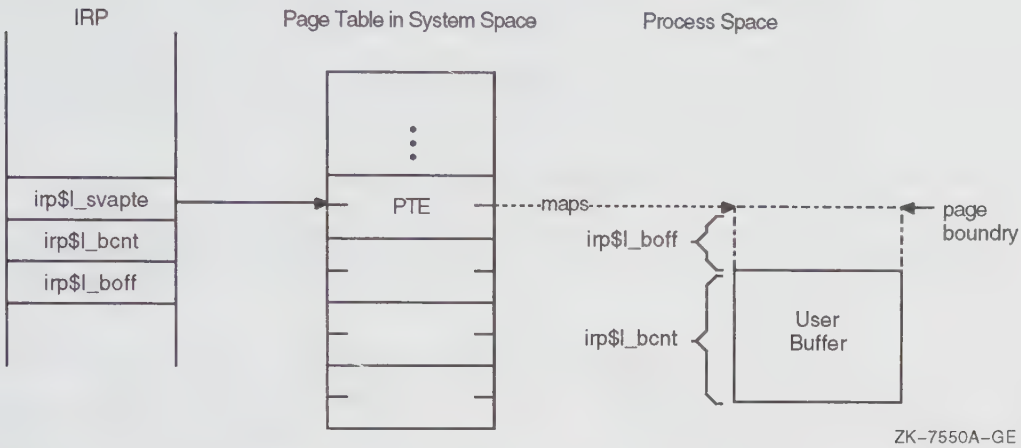
A driver executes mostly at elevated IPL with minimal context; it cannot request system services. This make it difficult to implement complex functions in drivers. An ancillary control process (ACP) is a separate thread of execution running in process context that implements complex driver functions. A driver's FDT routine passes an I/O request either to an ACP or to a driver's start I/O routine, depending on the I/O function requested. A complex function request such as opening a disk file or establishing a network logical link, for example is typically handled by an ACP.

The operating system provides two standard FDT routines that are applicable for direct I/O operations: EXE_STD$READ and EXE_STD$WRITE. When called by the driver, these routines completely prepare a direct I/O read or write request. Thus, a driver that uses these routines eliminates the need for its own device-specific FDT routines.

EXE_STD$READ and EXE_STD$WRITE are described in Chapter 19.

Figure 6–1 describes the association of a user buffer with an IRP.

Figure 6–1 Mapping the User Buffer for a Direct-I/O Function



6.6 FDT Routines for System Buffered I/O

Device drivers for buffered I/O operations generally contain their own device-specific FDT routines.

An FDT routine for a buffered I/O data transfer operation should confirm either read or write access to the user's buffer and allocate a buffer in system space. Sections 6.6.1 and 6.6.2 describe these tasks.

An FDT routine for a buffered I/O operation that does not involve data transfer accesses the function-dependent parameters of the $QIO request (`p1` to `p6`) from `irp$l_qio_p`*n*, where *n* is a parameter number from one to six. It performs any necessary preprocessing and uses one of the exit methods listed in Section 6.4.2.

6.6.1 Checking Accessibility of the User's Buffer

First the FDT routine invokes the CALL_READCHK or CALL_WRITECHK macros (which call EXE_STD$READCHK or EXE_STD$WRITECHK, respectively) to confirm write or read access to the user's buffer. Both of these routines write the transfer byte count into `irp$l_bcnt`. EXE_STD$READCHK also sets `irp$v_func` in `irp$l_sts` to indicate that it is a read function.

6.6.2 Allocating the System Buffer

Next, the FDT routine allocates a system buffer in the following manner:

1. It adds 12 bytes to the byte count passed in the p2 parameter of the user's I/O request (obtained from irp$l_qio_p2), thus accommodating the standard size of a system buffer header. This is the total system buffer size.

   Define the system buffer header.

   When you allocate the buffer, add the space you think you need and overhead for the system buffer header, which is 12. Here's the way you code it:

```
typedef struct _sysbuf_hdr {
    char *pkt_datap;            /* Pointer to start of data in packet */
    char *usr_bufp;             /* User VA of user buffer */
    short pkt_size;             /* Size of the system buffer packet * /
    short :16;
} SYSBUF_HDR;
```

2. It calls EXE_STD$DEBIT_BYTCNT_ALO to ensure that the job has sufficient remaining byte count quota to allow its use of the requested buffer. If the job has sufficient quota, EXE_STD$DEBIT_BYTCNT_ALO allocates the requested buffer from nonpaged pool, writes the buffer's size and type into its third longword, and subtracts the system buffer size from jib$l_bytcnt.

   The operating system also supplies the following routines, which perform the same type of work as EXE_STD$DEBIT_BYTCNT_ALO:

   EXE_STD$DEBIT_BYTCNT_BYTLM_ALO
   EXE_STD$DEBIT_BYTCNT(_NW)
   EXE_STD$DEBIT_BYTCNT_BYTLM(_NW)
   EXE_STD$ALLOCBUF

   These routines are available in the exe_routines.h header file in SYS$LIBRARY:SYS$LIB_C.TLB.

Once the buffer is allocated, the FDT routine takes the following steps:

1. Connects the buffer with the IRP.
2. Loads the address of the system buffer into irp$l_svapte.
3. Loads the total size of the system buffer into irp$l_boff.

4. Stores the starting address of the system buffer data area in the system buffer header and in the packet data pointer cell.

```
sys_bufp-> pkt_datap = (char*) sys_bufp + sizeof (sysbuf_hdr) ;
```

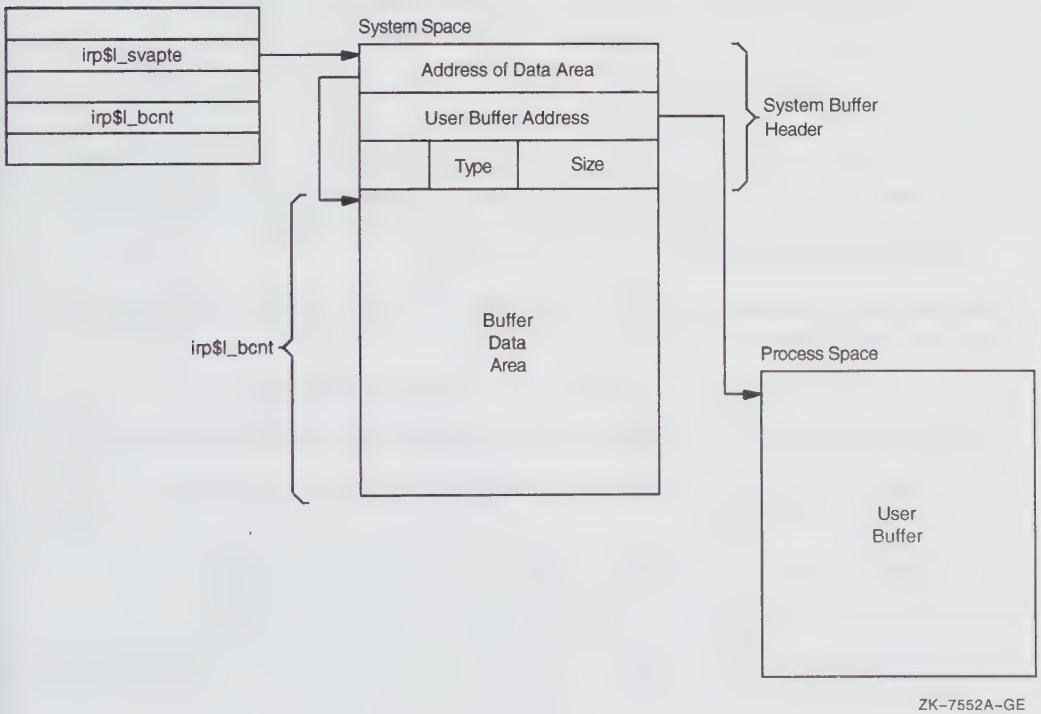
5. Stores the user's buffer address in the second longword of the header.

```
sys_bufp-> usr_bufp = irp->irp$l_qio_p1 ;
```

6. Copies data from the user buffer to the system buffer if the I/O request is a write operation.

At this point, the buffers are ready for the transfer. Figure 6–2 illustrates the format of the system buffer.

Figure 6–2 Format of System Buffer for a Buffered-I/O Read Function



6.6.3 Buffered-I/O Postprocessing

When the transfer finishes, the driver returns control to the operating system for completion of the I/O request. For example,

```
ioc_std$reqcom (SS$_NORMAL, 0, ucb);
```

The driver provides the two longword values for the I/O status block. The low-order 16 bits of the first longword must contain the final request status. The use of the upper 16 bits of the first longword and the entire second longword are driver-specific. Certain drivers use these fields to report a transfer byte count, for example.

The driver must leave the buffer header intact; I/O postprocessing relies on the header's accuracy. When system I/O postprocessing gains control, it performs three steps:

1. Calls EXE$CREDIT_BYTCNT to add the value in `irp$l_boff` to `jib$l_bytcnt`, thus updating the user's byte count quota.
2. If `irp$l_svapte` is nonzero, assumes a system buffer was allocated and checks to see whether `irp$v_func` is set in `irp$l_sts`.
3. If `irp$v_func` is clear, deallocates the system buffer used for the write operation; if `irp$v_func` is set, the special kernel-mode AST copies the data to the user's buffer and then deallocates the buffer in addition to performing other kernel-mode AST functions.

The special kernel-mode AST performs the following steps to complete a buffered read operation:

1. Obtains the address of the system buffer from `irp$l_svapte`.
2. Obtains the number of bytes to write to the user's buffer from `irp$l_bcnt`.
3. Obtains the address of the user's buffer from the second longword of the system buffer header.
4. Checks for write accessibility on all pages of the user's buffer.
5. Copies the data from the system buffer to the process buffer.
6. Deallocates the system buffer. Note that the system uses the size listed in the buffer's header to deallocate the buffer.

7

Writing a Start-I/O Routine

This chapter describes the general activities of a start-I/O routine for a typical device and provides a general overview of the simple fork and kernel process mechanisms. Your device driver start-I/O routine will need to use one of these mechanisms.

A device driver's start-I/O code path is the starting point for the system thread of execution that carries an I/O request through to completion. The start-I/O code path services I/O requests by interacting with the device controller. This code path activates a device and then waits for a device interrupt or timeout. The code path resumes after a device interrupt or timeout. It may activate the device again and wait for the next interrupt until all the device activity required by this I/O request has been completed. At that point the start-I/O code path completes by sending the I/O request on to the I/O postprocessing phase.

The processing performed by start-I/O code is device specific and usually contains elements that perform the following functions to activate:

- Analyzing the I/O function
- Transferring the details of a request from the IRP into the UCB
- Obtaining and initializing the controller
- Modifying device registers to activate the device

The start-I/O code of a DMA device driver performs additional tasks to prepare the device for a DMA transfer prior to activating the device. These tasks include the following:

- Obtaining I/O adapter resources such as map registers
- Computing the starting address of a data transfer

7.1 Simple Forks and Kernel Processes

To create a start-I/O code path, you can use either a simple fork or a kernel process mechanism. Which mechanism you choose depends on the complexity of your driver and how much context you want preserved when you suspend execution.

In the simple fork mechanism, the execution model is a chain of simple routines. Three fork parameters are preserved across stalls. Any other driver state must be maintained in nonpaged pool structures across stalls. When you create a fork, you specify a routine that will execute at a future time but your current thread of execution continues as well. If you wish to suspend execution until that fork thread is invoked you must perform no more additional processing except to return out of all the routines invoked by the current thread back to its caller. Any local variables that the current thread had will thus be deallocated and will not be available to the fork thread when it resumes.

In the kernel process mechanism, the execution model is a single routine at the top of a deep call tree. The execution can be stalled and resumed almost anywhere within the call tree. The stack local storage, all nonscratch registers, and current procedure context is preserved across all the stalls. A kernel process is a simple fork thread with its own private kernel stack. Therefore, when you stall a kernel process for an interrupt or to fork to lower IPL your execution context is saved and later resumed using that same private kernel stack. This is how your current procedure context including the program counter and all your local variables are maintained.

The kernel process mechanism requires more system overhead than the simple fork mechanism because the kernel process mechanism utilizes the simple fork mechanism along with a Kernel Process Block (KPB) data structure in nonpaged pool and a private kernel stack. The kernel process mechanism must save and restore additional context when suspending and resuming execution. However, it's often much easier to write and maintain a complex driver by using the kernel process mechanism rather than the simple fork mechanism. For example the OpenVMS Alpha SCSI class and port drivers use the kernel process mechanism. In contrast, the vastly simpler parallel printer port driver uses the simple fork mechanism.

7.2 Context of Driver Start-I/O Code

A start-I/O routine does not run in the context of a user process. Rather, it has the following context:

System context	Driver code can only refer to system virtual addresses.
Kernel mode	Execution occurs in the most privileged access mode and can, therefore, change IPL and obtain spinlocks.
High IPL	The system routine that creates a driver fork process obtains the driver's fork lock, raising IPL to driver fork level before activating the driver.
Kernel stack	Execution occurs on the kernel stack.
IRP	A pointer to the current I/O request packet.
UCB	A pointer to the unit control block for the device that the I/O request has been issued to.

The system I/O request packet-queuing routines invoke the driver start-I/O routine after copying the following IRP fields into their corresponding slots in the device's UCB:

- `irp$l_bcnt` → `ucb$l_bcnt`
- `irp$l_boff` → `ucb$l_boff`
- `irp$l_svapte` → `ucb$l_svapte`

7.3 Simple Fork Start-I/O Interface

If a driver is using the simple fork mechanism, the driver's start-I/O routine is invoked in a fork thread using the following interface:

```
void driver_startio (IRP *irp, UCB *ucb)
```

- The `irp` parameter contains the address of the I/O request packet.
- The `ucb` parameter contains the address of the unit control block.

More than likely the fork thread created to start this I/O request will end before all the operations required to carry out this I/O request are completed. Subsequent fork threads will be queued, run, and end, until the I/O request is completed. Each fork thread requires an individual routine.

For more details about how to code start-I/O routines using the simple fork mechanism, see Chapter 8.

7.4 Kernel Process Start-I/O Interface

If a driver is using the kernel process mechanism, the driver's start-I/O routine is invoked in the context of a kernel process using the following interface:

```
void driver_kpstartio (KPB *kpb)
```

- The kpb parameter contains the address of the kernel process block (KPB) that was created for this I/O request. The IRP and UCB pointers for the current I/O request are obtained from the KPB.

This kernel process will exist until the I/O request is completed and control is returned back to the caller of the driver's start-I/O routine. However, this kernel process may suspend execution any number of times before it returns and ends. The entire start-I/O code path is logically contained in the single kernel process start I/O routine. Of course, this single start-I/O routine can call out to other routines which in turn might suspend the kernel process or call other routines.

For more details about how to code start-I/O routines using the kernel process mechanism, see Chapter 9.

7.5 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.

8

Using the Simple Fork Start-I/O Mechanism

This chapter describes the routines to include if your driver is using the simple fork mechanism for its start-I/O code path.

8.1 Overview of Simple Fork Process Routines

Some or all of the following routines may need to be included in the simple fork start-I/O code path of your driver:

- Start-I/O entry routine

  ```
  void driver_startio (IRP *irp, UCB *ucb);
  ```

  This routine is required in all but the most trivial drivers.

- Channel grant routine

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

  This routine is required only if a device controller is shared by multiple device units.

- Counted resource grant routine

  ```
  void driver_res_grant (CRAB *crab, CRCTX *crctx,
                         int64 ctx1, int64 ctx2, int64 ctx3,
                         int status);
  ```

  This routine is required if you need to allocate adapter map registers or other resources.

- Interrupt service routine

  ```
  void driver_isr (IDB *idb);
  ```

  This routine is usually required if the driver is handling a physical device (in contrast to a virtual device).

- Resume from interrupt routine

  ```
  void driver_resume_fi (IRP *irp, int64 fr4, UCB *ucb);
  ```

  This routine may be needed if the driver has an interrupt service routine.

- Interrupt timeout routine

  ```
  void driver_timeout (IRP *irp, int64 fr4, UCB *ucb);
  ```

  This routine is required if the driver has an interrupt service routine.

- Fork routines

  ```
  void driver_fork (IRP *irp, int64 fr4, UCB *ucb);
  ```

  A driver may use an arbitrary number of fork routines on its start I/O code path.

8.1.1 Transferring Control to the Start-I/O Routine

The start-I/O code path routine of a device driver using the simple fork mechanism gains control from either of two system routines: EXE_STD$QIODRVPKT or IOC_STD$REQCOM.

When FDT processing is complete for an I/O request, the FDT routine transfers control to EXE_STD$QIODRVPKT, which, in turn, calls EXE_STD$INSIOQ. If the designated device is idle, EXE_STD$INSIOQ calls IOC_STD$INITIATE to create a driver fork process. The driver fork process then gains control in the start-I/O routine of the appropriate driver. If the device is busy, EXE_STD$INSIOQ queues the packet to the device unit's pending-I/O queue.

After a device completes an I/O operation, the driver fork process exits by transferring control to IOC_STD$REQCOM. IOC_STD$REQCOM inserts the I/O request packet (IRP) for the finished transfer into the postprocessing queue. It then dequeues the next IRP from the device unit's pending-I/O queue and calls IOC_STD$INITIATE to initiate the processing of this I/O request in the driver's fork process at the entry point of the driver's start-I/O routine. The driver's start I/O routine is invoked at driver fork IPL with the driver fork spinlock held using the following interface:

```
void driver_startio (IRP *irp, UCB *ucb)
```

- The irp parameter contains the address of the I/O request packet.
- The ucb parameter contains the address of the unit control block.

8.1.2 Obtaining Controller Access

If the device is one of several attached to a controller, the start-I/O routine uses the IOC_STD$PRIMITIVE_REQCHANH or IOC_STD$PRIMITIVE_REQCHANL routines to dynamically assign the controller's data channel to the device unit. Controllers that control only one device do not require arbitration for the controller's data channel. For the convenience of the interrupt service routine, such controllers are often permanently assigned to their single unit in the unit initialization routine.

```
idb->idb$ps_owner = ucb;
```

The transfer being controlled by the start-I/O routine discussed here requires no seek preceding the transfer. Therefore all requests for the controller are done by using IOC_STD$PRIMITIVE_REQCHANL which queues them at the end of the controller wait queue. Disk I/O is an example of a transfer that requires a seek first. To permit seeks to be overlapped with transfers, you can use the IOC_STD$PRIMITIVE_REQCHANH routine to insert seek requests for a channel at the head of the channel wait queue.

If the channel is not available, IOC_STD$PRIMITIVE_REQCHANL returns an error status. When the channel is eventually released and granted to this requestor the channel grant routine is invoked.

```
/* Request the controller */

ucb->ucb$l_fpc = driver_chn_grant;
status = ioc_std$primitive_reqchanl (irp, ucb, &idb);

/* Return if controller was not granted.  Routine driver_chn_grant will
 * be eventually called when we get the controller.
 */

if ( ! $VMS_STATUS_SUCCESS(status) ) return;

/* Controller was granted, continue processing */
```

IOC_STD$PRIMITIVE_REQCHANL also writes the address of the new channel-owner's UCB in the owner field of the IDB (`idb$l_owner`). The driver's interrupt service routine later reads this IDB field to determine which device unit owns the controller's data channel. A driver for a single-unit controller must fill the `idb$l_owner` field in its controller or unit initialization routines.

8.1.3 Obtaining and Converting the I/O Function Code and Its Modifiers

The start-I/O routine extracts the I/O function code and function modifiers from the field `irp$l_func` and translates them into device-specific function codes, which it loads into the device's CSR or other control registers. The start-I/O routine creates and modifies a bit mask that is to be loaded into the CSR when the driver starts the device. To accomplish this, the start-I/O routine converts the function modifiers contained in `irp$l_func` into device-specific bit settings in the general register.

8.1.4 Preparing the Device Activation Bit Mask

For a typical device, the start-I/O routine prepares the device-activation bit mask by setting the interrupt-enable bit and the go bit in the general purpose register that also contains the high-order bits of the bus address and the device-function bits. At this point, the general register contains a complete command for starting the transfer, also known as the **control mask**.

When the start-I/O routine copies the contents of the register into the device's CSR, the device starts the transfer. Before activating the device, however, the start-I/O routine should perform the steps described in Sections 8.1.5 and 8.1.6.

8.1.5 Synchronizing Access to the Device Database

The start-I/O routine invokes the system macro device_lock to synchronize its access to device registers with the interrupt service routine. This macro invocation is doubly important, for it establishes the context wherein the driver can later issue the wait-for-interrupt macro (`wfikpch` or `wfirlch`). The wait-for-interrupt macros expect the driver's fork IPL value as obtained conviently by the device_lock macro. In addition, the wait-for-interrupt macros issue the device_unlock macro to release ownership of the device lock and restore the previous IPL.

```
int orig_ipl;          /* Place to save original IPL */

/* Obtain device lock, saving original IPL */

device_lock (ucb->ucb$l_dlck, RAISE_IPL, &orig_ipl);

/* Touch device registers and activate the device */

...

/* Setup to wait for the interrupt */

wfikpch (rfi_routine,     /* Driver resume-from-interrupt routine */
         tmo_routine,     /* Driver interrupt timeout routine */
         irp, 0, ucb,     /* Paramters for the above 2 routines */
         TMO_SECONDS,     /* Timeout value in seconds */
         orig_ipl);       /* Fork IPL to return to */
```

```
return;
```

8.1.6 Checking for a Local Processor Power Failure

After synchronizing access to device registers, the start-I/O routine invokes the `setipl` macro to raise IPL to IPL$_POWER to block all interrupts on the local processor.

The start-I/O routine then examines the power failure bit in the UCB's status longword (`ucb$v_power` in `ucb$l_sts`) to determine whether a local power failure has occurred since the start-I/O routine gained control. If the bit is not set, the transfer can proceed.

If the bit is set, a power failure might have occurred between the time that the start-I/O routine wrote the first device register and the time that the start-I/O routine is ready to activate the device. Such a power failure could modify the already-written device registers and cause unpredictable device behavior if the device were to be started.

If the bit `ucb$v_power` is set, the start-I/O routine branches to an error handler in the driver. The driver error handler must perform the following actions:

- Clear `ucb$v_power`
- Issue the `device_unlock` macro to release the device lock and restore IPL to fork IPL

After performing these tasks, many drivers transfer control to the beginning of the start-I/O routine, which restarts the processing of the I/O request.

8.1.7 Activating the Device

If no power failure has occurred, the start-I/O routine copies the contents of the control mask into the device's CSR. When the device notices the new contents of the device register, it begins to transfer the requested data.

8.2 Waiting for an Interrupt or Timeout

Once the start-I/O routine activates the device, the driver fork process cannot proceed until one of these events occurs:

- The device generates a hardware interrupt.
- The device does not generate a hardware interrupt within an expected time limit, which is to say that a device timeout occurs.

Still executing at IPL$_POWER, the driver's start-I/O routine asks the operating system to suspend the driver fork process by invoking one of the following macros:

`wfikpch`	Wait for an interrupt or timeout and keep the controller data channel
`wfirlch`	Wait for an interrupt or timeout and release the controller data channel

Both macros invoke routines that release ownership of the device lock, relinquish synchronization, and return IPL to the previous level when exiting.

Drivers generally keep the controller data channel while waiting for the interrupt or timeout. Drivers of devices with dedicated controllers always keep the channel because only one unit ever needs it. For devices that share a controller, some operations, such as disk seeks, do not require the controller once the operation has begun. In such cases, the driver can release the controller's data channel while waiting for an interrupt or timeout so that other units on the controller can start their operations.

8.3 Writing an Interrupt Service Routine

When a device generates a hardware interrupt, it requests an interrupt at the appropriate device interrupt priority level (IPL). Either the device or its adapter requests a processor interrupt at that IPL. When the processor executes at an IPL below that device IPL, interrupt dispatching begins.

The mechanism of interrupt dispatching has no direct bearing on the contents of a driver's interrupt service routine. Its implementation varies according to the Alpha system and I/O subsystem in use.

For most device drivers, the driver prologue table (DPT) contains, in the reinitialization section established by the `dpt_store_isr` macro, the address of one or more interrupt service routines. Each interrupt service routine corresponds to an interrupt vector on the I/O bus. You specify the interrupt vector using the SYSMAN command IO CONNECT.

Most device interrupt service routines perform the following functions:

- Locate the device's unit control block (UCB)
- Determine whether the interrupt was solicited
- Reject or process unsolicited interrupts
- Activate the suspended driver to process solicited interrupts

The remaining sections of this chapter describe the handling of solicited and unsolicited interrupts in further detail.

8.3.1 Servicing a Solicited Interrupt

When a driver's fork process activates a device and expects to service a device interrupt as a result, the fork process suspends its execution and waits for an interrupt to occur. The suspended driver is represented only by the contents of the fork block in the device's UCB, which contain the following information:

- The pointer to the IRP in ucb$q_fr3
- An arbitrary second fork routine parameter in ucb$q_fr4
- The address of the resume from interrupt routine in ucb$l_fpc
- The address of the timeout handling routine in ucb$ps_toutrout

The interrupt service routine is called by the system interrupt dispatcher using the following interface:

```
void driver_isr (IDB *idb);
```

A driver's interrupt service routine performs the following tasks to process the interrupt and transfer control to the waiting driver:

1. Obtains the address of the device's UCB from the IDB, as follows:

   ```
   ucb = idb->idb$l_owner;
   ```

2. Issues the device_lock macro to obtain synchronized access to device registers. However, there is no need to raise or save the original IPL because we know we are at device IPL in the interrupt service routine.

   ```
   device_lock (ucb->ucb$l_dlck, NORAISE_IPL, NOSAVE_IPL);
   ```

3. Tests the interrupt-expected bit in the UCB status longword (ucb$v_int in ucb$l_sts). If the bit is set, the driver is waiting for an interrupt from this device. After performing this test, the interrupt service routine *must* clear ucb$v_int to indicate that it has received the expected interrupt.

   ```
   if ( ! ucb->ucb$v_int ) {          /* Dismiss unexpected interrupt */
       device_unlock (ucb->ucb$l_dlck, NOLOWER_IPL, SMP_RESTORE);
       return;
   }
   ucb->ucb$v_int = 0;                /* This interrupt being handled */
   ```

Note

Because device timeout processing mostly occurs at fork IPL (see Section 8.5), a driver's interrupt service routine, executing at device IPL, could interrupt the processing of a timeout on the same device unit. For this reason, the driver's interrupt service routine should check the interrupt-expected bit (`ucb$v_int`) before handling the interrupt. The operating system clears this bit before it calls the driver's timeout handler.

4. Obtains device-status or controller-status information from the device registers, if necessary, and stores the status information in the UCB.
5. Issues the `rfi` macro to invoke the driver resume from interrupt routine

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

The restored driver should execute as briefly as possible in interrupt context. As soon as possible, the driver should invoke the `iofork` macro to request the creation of a fork process at the driver's fork IPL in order to complete the I/O operation. Forking lowers the IPL of driver execution below device IPL, allowing the processor to service additional device interrupts. The `iofork` macro calls the routine EXE_STD$PRIMITIVE_FORK. EXE_STD$PRIMITIVE_FORK inserts into the appropriate fork queue the UCB fork block that describes the driver process. It then returns control to the driver's interrupt service routine. (See Section 8.4.2 for additional information on driver forking.)

The interrupt service routine then performs the following steps:

1. Issues the `device_unlock` macro to release ownership of the device lock

```
device_unlock (ucb->ucb$l_dlck, NOLOWER_IPL, SMP_RESTORE);
```

2. Returns to the interrupt dispatcher
3. The interrupt dispatcher dismisses the interrupt

8.3.2 Servicing an Unsolicited Interrupt

A device requests an interrupt to indicate to a driver that the device has changed status. If a driver's fork process starts an I/O operation on a device, the driver expects to receive an interrupt from the device when the I/O operation completes or an error occurs.

Other changes in the device's status occur when the device has not been activated by a device driver. The device reports such a change by requesting an unsolicited interrupt. For example, when a user types on a terminal, the terminal requests an interrupt that is handled by the terminal driver. If the terminal is not attached to a process, the terminal driver causes the login procedure to be invoked for the user at the terminal.

As another example, an unsolicited interrupt occurs whenever a disk drive goes off line, as could happen when an operator spins it down or pushes the offline button. The disk driver services the interrupt by altering volume and unit status bits in the disk device's UCB.

Devices request unsolicited interrupts because some external event has changed the status of the device. A device driver can handle these interrupts in two ways:

- Ignore the interrupt as spurious
- Examine the device registers and take action according to their indications of changed status, and then poll for any other changes in device status

As mentioned in Section 8.3.1, an interrupt service routine first obtains the address of the device's UCB from the IDB. It then issues the `device_lock` macro to obtain synchronized access to device registers.

The routine determines whether an interrupt is solicited or not by examining the interrupt-expected bit in the UCB status longword (`ucb$v_int` in `ucb$l_sts`).

If the driver decides to handle the unsolicited interrupt, it must observe certain precautions. Certain methods of servicing unsolicited interrupts—for instance sending a message to the operator or the job controller's mailbox—must be accomplished at an IPL lower than device IPL. Although the interrupt service routine can legitimately fork to accommodate unsolicited interrupts, it should exercise extreme caution in doing so.

If `ucb$v_bsy` is set in `ucb$l_sts`, the UCB fork block is currently in use by the driver's start-I/O routine. An attempt by the interrupt service routine to concurrently use the fork block can destroy the fork context already stored in that UCB. Moreover, if `ucb$v_bsy` is not set, the interrupt service routine

cannot safely assume that the fork block is not in use, for it may be currently employed to service a previous unsolicited interrupt.

To avoid confusion, code servicing an unsolicited interrupt must ensure that the fork block it requires is not being used. Perhaps the safest method to guarantee this is for the driver to define a separate fork block in a device-specific UCB extension. The driver should also define a semaphore to control access to this fork block and protect against multiple forking. Note that the driver should access the semaphore using atomic builtins.

If, upon servicing an unsolicited interrupt, the driver's interrupt service routine examines the semaphore and discovers that a fork is already in progress, it should not attempt to fork.

The system routine that creates the fork process (once these conditions are satisfied) returns control to the interrupt service routine. The interrupt service routine then releases the device lock, and returns to dismiss the interrupt.

8.4 Completing an I/O Request and Handling Timeouts

Once a driver has activated the device and invoked the wait-for-interrupt macro, the driver remains suspended until the device requests an interrupt or times out.

If the device requests an interrupt, the driver's interrupt service routine handles the interrupt and then reactivates the driver at the instruction following the wait-for-interrupt macro. The reactivated driver performs device-dependent I/O postprocessing.

If the device does not request an interrupt within the designated time interval, the system transfers control to the driver's timeout handling routine. The address of the timeout handling routine is specified to the wait-for-interrupt macros.

8.4.1 I/O Postprocessing

Once the driver interrupt service routine has processed an interrupt, it transfers control to the driver resume from interrupt routine. At this point, the driver is executing in interrupt context. If the driver were to continue executing in interrupt context, it would lock out most other processing on the processor including the handling of hardware interrupts.

To restore the driver to the context of a driver fork process, the driver invokes the `iofork` macro. Once the fork process has been created and dispatched for execution, it executes the driver code that completes the processing of the I/O request.

8.4.2 IOFORK

The `iofork` macro is used to transition the interrupt service routine context to that of a fork process by performing the following steps:

1. It disables software timeouts by clearing the timeout enable bit in the UCB status longword (`ucb$v_tim` in `ucb$l_sts`).
2. It saves the address of the fork routine in the UCB fork block (`ucb$l_fpc`).
3. It saves the two fork parameters of the current driver context in the UCB fork block (`ucb$q_fr3` and `ucb$q_fr4`).
4. It obtains the fork lock index of the driver from the UCB (`ucb$b_flck`) and uses it to determine in which fork queue it should place the fork block.
5. It inserts the address of the UCB fork block into the processor-specific fork queue corresponding to the driver's fork IPL.
6. Finally, if the fork block is the first entry in the fork queue, EXE_STD$PRIMITIVE_FORK requests a software interrupt from the local processor at the driver's fork IPL.

The steps listed previously move the fork process context into the UCB's fork block. The driver's fork process resumes processing when the system fork dispatcher dequeues the UCB fork block from the fork queue, and reactivates the driver at the driver's fork IPL.

8.4.3 Completing an I/O Request

When the operating system reactivates a driver's fork process by dequeuing the fork block, the driver resumes processing of the I/O operation holding the appropriate fork lock at fork IPL.

1. Releases map registers
2. Releases the controller (applies only to drivers of devices on multiunit controllers)
3. Checks device register images saved in the UCB to determine the status of the I/O operation
4. Saves in the I/O reuest packet (IRP) the status code, transfer count, and device-dependent status that is to be returned to the user process in an I/O status block (IOSB)
5. Returns control to the operating system

8.4.3.1 Releasing the Controller

To release the controller channel, the driver calls the IOC_STD$RELCHAN routine. If another driver is waiting for the controller channel, IOC_STD$RELCHAN queues that driver's fork process which will invoke its controller grant routine.

Drivers for devices with dedicated controllers need not release the controller's data channel. By means of code in the unit initialization routine, these drivers set up the device's UCB so that the device owns the controller permanently.

Drivers must be executing at driver's fork IPL when they call IOC_STD$RELCHAN.

8.4.3.2 Saving Status, Count, and Device-Dependent Status

To save the status code, transfer count, and device-dependent status, the driver performs the following steps:

1. Loads a success status code (SS$_NORMAL), or whatever is appropriate, into the low-order 16-bits of the first I/O status longword.

```
int iost1;
iost1 = SS$_NORMAL;
```

2. Loads the number of bytes transferred into the high-order 16 bits of the first I/O status longword, if the I/O operation performed by the device is a transfer function.

```
int iost1;
iost1 = iost1 & (byte_count<<16);
```

3. Loads device-dependent status information, if any, into the second I/O status longword.

```
int iost2;
iost1 = dev_devpend;
```

8.4.3.3 Completing the I/O Request

Finally, the driver fork process completes the I/O request by calling routine IOC_STD$REQCOM.

```
ioc_std$reqcom (iost1, iost2, ucb);
```

IOC_STD$REQCOM locates the address of the I/O request packet (IRP) corresponding to the I/O operation in the device's UCB (`ucb$l_irp`). It then writes the two longwords of completion status contained into `irp$l_iost1` and `irp$l_iost2`. These two longwords will eventually be returned to the user process in the I/O status block specified in the original $QIO system service.

IOC_STD$REQCOM then inserts the IRP in the local processor's I/O-postprocessing queue and requests a software interrupt at IPL$_IOPOST from the current processor so the postprocessing begins when IPL drops below IPL$_IOPOST.

If the error-logging bit is set in the device's UCB (ucb$v_erlogip in ucb$l_sts), IOC_STD$REQCOM obtains the address of the error message buffer from the UCB ucb$l_emb. It then writes the following information into the error buffer:

- Final device status (UCB_DEVSTS)
- Final error count ucb$b_ertcnt
- Maximum error retry count for the driver
- Two longwords of completion status

To release the error message buffer, IOC_STD$REQCOM calls ERL$RELEASEMB.

If any IRPs are waiting for driver processing, IOC_STD$REQCOM dequeues an IRP from the head of the queue of packets waiting for the device unit (ucb$l_ioqfl), and transfers control to IOC_STD$INITIATE. IOC_STD$INITIATE initiates execution of this I/O request in the driver's fork process, by activating the driver's start-I/O routine.

Otherwise, IOC_STD$REQCOM clears the unit-busy bit in the device's UCB status longword (ucb$v_bsy in ucb$l_sts) and transfers control to IOC_STD$RELCHAN to release the controller channel in case the driver failed to do so.

The remaining steps in processing the I/O request are performed by system I/O postprocessing.

8.5 Timeout Handling Routines

The operating system transfers control to the driver's timeout handling routine if a device unit does not request an interrupt within the time limit specified in the invocation of the wait-for-interrupt macro. Among its other activities, the system software timer fork routine running at IPL$_SYNCH, scans UCBs once every second to determine whether a device has timed out.

When the software timer interrupt service routine locates a device that has timed out, the routine calls the driver's timeout handling routine by performing the following steps:

1. It obtains both the fork lock and the device lock associated with the device unit to synchronize access to its fork database and device database. It raises IPL to device IPL as a result of obtaining the device lock.
2. It raises IPL on the local processor to IPL$_POWER to block local power failure servicing.
3. It disables expected interrupts and timeouts on the device by clearing bits in the status field of the device's UCB (ucb$v_int and ucb$v_tim in ucb$l_sts).
4. It sets the device-timeout bit in the UCB status field (ucb$v_timout in ucb$l_sts).
5. It lowers IPL to hardware device interrupt IPL (ucb$b_dipl).
6. It transfers control to the driver's timeout handling routine, which is contained in ucb$ps_toutrout.

The driver's timeout handling routine executes in the following context:

- Only system address space may be accessed.
- The processor is running in kernel mode.
- The processor is running on the kernel stack.
- The processor holds both fork lock and device lock.
- IPL is at hardware device interrupt level.

Certain timeout handling routines may find it useful to fork to execute low priority code or to accomplish certain tasks, such as the restarting of an I/O request. If a driver uses this method, its interrupt service routine should check the interrupt-expected bit (ucb$v_int) before handling the interrupt. The operating system clears this bit before it calls the driver's timeout handling routine. This allows the routine to determine whether device-timeout processing is in progress at fork IPL.

During recovery from a power failure, the operating system forces a device timeout by altering the timeout field (ucb$l_duetim) of a UCB if that device's UCB records that the unit is waiting for an interrupt or timeout (ucb$v_int and ucb$v_tim set in ucb$l_sts). The timeout handling routine can perceive that recovery from a power failure is occurring by examining the power bit (ucb$v_power in ucb$l_sts) in the UCB.

A timeout handling routine usually performs one of three functions:

- It retries the I/O operation unless a retry count is exhausted.

- It aborts the I/O request, returning status (for instance, SS$_TIMEOUT).
- It sends a message to an operator mailbox and waits for a subsequent interrupt or timeout.

9

Using the Kernel Process Start-I/O Mechanism

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.

9.1 Kernel Process Data Structures

Each kernel process has two data structures:

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

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.

- Spinlock area

  The spinlock 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 spinlock 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 spinlock 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 9–1 `kpb$is_stack_size`, `kpb$ps_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 9–1 shows the stack and the fields in the KPB related to it.

9.2 Kernel Process Routines

The kernel process routines that create a kernel process and its associated structures and maintain the kernel process environment are divided into four basic areas:

- Allocation/deallocation
- Start/stop
- Stall/restart
- Miscellaneous support

Table 9–1 summarizes these routines. The remaining sections in this chapter describe how to use kernel process system routines to set up and use a driver kernel process.

Figure 9–1 Kernel Process Private Stack

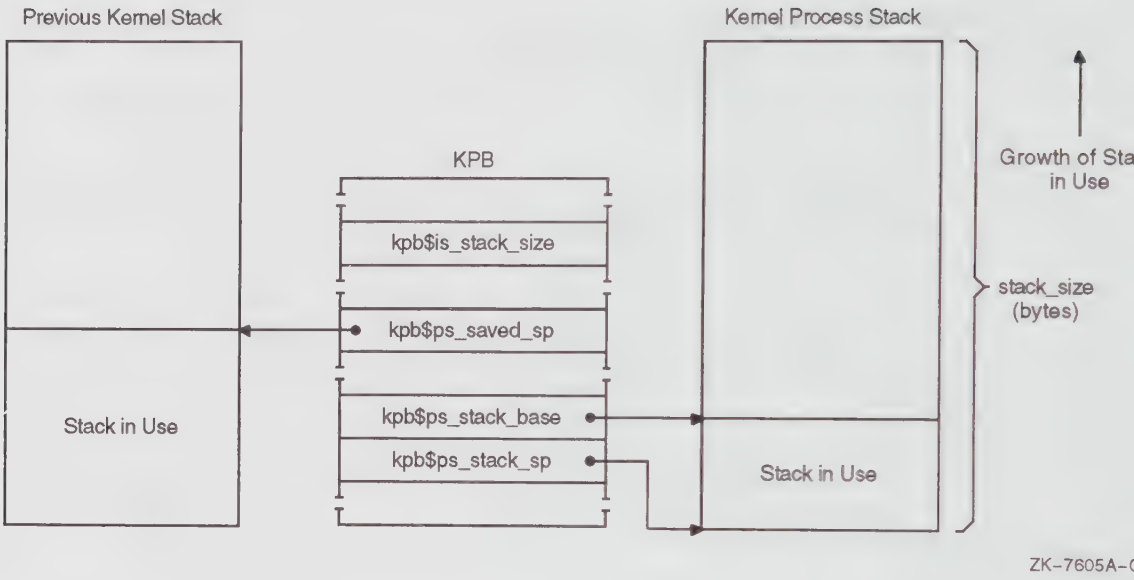


Table 9–1 System Routines That Create and Manage Kernel Processes

System Routine	Function
EXE_STD$KP_STARTIO	Allocates and sets up a KPB and a kernel process private stack, and starts up the execution of a kernel process used by a device driver
EXE$KP_ALLOCATE_KPB	Allocates a KPB and its kernel process private stack
EXE$KP_START	Starts the execution of a kernel process
EXE$KP_STALL_GENERAL	Stalls the execution of a kernel process
EXE$KP_FORK	Stalls a kernel process in such a manner that it can be resumed by the OpenVMS fork dispatcher
EXE$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
IOC$KP_REQCHAN	Stalls a kernel process in such a manner that it can be resumed by the granting of a device controller channel
IOC$KP_WFIKPCH IOC$KP_WFIRLCH	Stalls a kernel process in such a manner that it can be resumed by device interrupt processing
EXE$KP_RESTART	Resumes the execution of a kernel process
EXE$KP_END	Terminates the execution of a kernel process
EXE$KP_DEALLOCATE_KPB	Deallocates a KPB and its kernel process private stack

9.3 Creating a Driver Kernel Process

A driver that specifies EXE_STD$KP_STARTIO in its table initilization routine creates a kernel process in which its own start-I/O routine runs. For example, using the following macros in the `driver$init_tables` routine is sufficient to set up a kernel process for most drivers and start execution of the driver's start-I/O routine as a kernel process thread:

```
ini_ddt_start      (&driver$ddt, exe_std$kp_startio);
ini_ddt_kp_startio (&driver$ddt, driver_kpstartio);
```

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 second parameter to the `ini_ddt_kp_startio` macro.

EXE_STD$KP_STARTIO customizes the kernel process environment specifically for driver kernel processes and 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 8 KB). A driver contributes a value to ddt$is_stack_bcnt by using the table-building macro ini_ddt_kp_stack_size.
- Specifies irp$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 spinlock sections and indicates that the KPB should be deleted when the kernel process is terminated.
- Calls 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 the 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. A driver contributes a value to ddt$is_reg_mask by specifying the ini_ddt_kp_reg_mask macro in the driver's table initilization routine. EXE_STD$KP_STARTIO excludes any registers that are illegal in a kernel process register save mask 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 ini_ddt_kp_startio macro in the driver's table initilization routine.
- The driver's start I/O routine 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:

```
void driver_kpstartio (KPB *kpb) {
    IRP *irp; UCB *ucb;
    char data[32];
    .
    irp = kpb->kpb$ps_irp;
    ucb = kpb->kpb$ps_ucb;
    .
    ioc$kp_wfikpch (kpb, TMO, newipl);
    .
    ioc$kp_fork (kpb, ucb);
    .
    ioc_std$reqcom (SS$_NORMAL, 0, ucb);
```

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$KP_STARTIO in the `ini_ddt_start` macro. Rather it must make explicit calls to EXE$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 9.6 for additional information about using the KPB parameter area.

For the routines that manipulate kernel process structures, such as EXE$KP_ALLOCATE_KPB and EXE$KP_START, a driver should check the status value and take appropriate action.

9.4 Suspending a Kernel Process

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) 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.

9.5 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 simply returning from the start I/O routine.

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`.

Note that the completion of the I/O request must be indicated by an explicit call to the EXE_STD$REQCOM routine before returning and ending the kernel process.

9.6 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_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:

```
/* Layout of our driver-specific KPB parameter area */

typedef struct _my_kpb_param {
    int kpb_param1;
    int kpb_param2;
} MY_KPB_PARAM;

KPB *kpb;                   /* Pointer to allocated KPB */
MY_KPB_PARAM *kpb_prm;      /* Pointer to param area in KPB */

/* Allocate a KPB with a parameter area */

status = exe$kp_allocate_kpb (&kpb,
                              MY_KP_STACK_SIZE,
                              (KP$M_IO & KP$M_DEALLOC_AT_END),
                              sizeof(MY_KPB_PARAM) );
if ( ! $VMS_STATUS_SUCCESS(status) ) return status;
```

```
/* Copy some values into the KPB parameter area */

kpb_prm = (MY_KPB_PARAM *) kpb->kpb$ps_prm_ptr;
kpb_prm->kpb_param1 = VALUE1;
kpb_prm->kpb_param2 = VALUE2;

/* Start the kernel process, executing the kp_routine */

status = exe$kp_start (kpb,kp_routine,KPREG$K_HLL_REG_MASK);
if ( ! $VMS_STATUS_SUCCESS(status) ) return status;
```

9.7 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 spinlocks and IPL). The initiator of the kernel process thread cannot assume that the kernel process has already executed when EXE$KP_START returns control.

10

Initializing a Device Driver

This chapter quickly summarizes the initialization sequence OpenVMS Alpha drivers follow, defines the tables every device driver contains, and describes driver initialization macros and routines.

10.1 Overview of the Driver Initialization Sequence

OpenVMS Alpha device drivers are initialized as follows:

1. Driver image is loaded.
2. If the driver image contains the global entry point `driver$init_tables`, the following routine is called:

   ```
   int driver$init_tables (void);
   ```

3. Driver tables are checked and used to guide remaining steps.
4. New I/O database structures created.
5. Driver's structure initialization routine is called:
   - Initializes fields in newly created UCB, DDB, CRB, IDB.

     ```
     void lr$struc_init (CRB *crb,..., LR_UCB *ucb) {
         ucb->ucb$r_ucb.ucb$b_flck = SPL$C_IOLOCK8;
     ```

   - Specified by DPT, for example:

     ```
     ini_dpt_struc_init (&driver$dpt, lr$struc_init );
     ```

6. Structure re-initialization routine is called.
   - Initializes fields in newly created UCB, DDB, CRB, IDB, when the driver image is loaded. For example:

     ```
     void lr$struc_reinit (CRB *crb,..., LR_UCB *ucb) {
         dpt_store_isr (crb, lr$interrupt);
     ```

- Specified by DPT, for example:

```
ini_dpt_struc_reinit (&driver$dpt, lr$struc_reinit );
```

7. I/O database structures are completed and linked in.
8. Driver CSR mapping routine is called.
   - New driver routine for mapping device registers, which reside in an I/O space, into system virtual address space.

```
int lr$csr_mapping (IDB *idb, DDB *ddb, CRB *crb);
```

   - Environment appropriate for calling ioc$map_io routine.

```
int ioc$map_io (ADP *adp, int node,
                uint64 *phys_offset, int num_bytes,
                int attr, uint64 *iohandle);
```

   - Specified by DDT, for example:

```
ini_ddt_csr_mapping (&driver$ddt, lr$csr_mapping);
```

9. Driver controller init routine is called.

```
int lr$ctrl_init (IDB *idb, DDB *ddb, CRB *crb);
```

   - May initialize device controller
   - Also called during power recovery
   - Specified by DDT, for example:

```
ini_ddt_ctrlinit (&driver$ddt, lr$ctrl_init );
```

10. Driver unit init routine is called.

```
int lr$unit_init (IDB *idb, LR_UCB *ucb);
```

   - May initialize device unit
   - Makes unit ready to accept I/O requests
   - Also called during power recovery
   - Specified by DDT, for example:

```
ini_ddt_unitinit (&driver$ddt, lr$unit_init );
```

The **unit initialization routine** and **controller initialization routine** prepare a device or controller for operation when the driver-loading procedure loads the driver into memory and when the system recovers from a power failure. The amount and type of initialization needed by devices and controllers vary according to the device type and the I/O bus to which the device or controller is attached.

10.2 Device Driver Tables

Every device driver contains three tables that describe the device and driver:

- **Driver prologue table (DPT)**—Describes the device type, driver name, and fields in the I/O database to be initialized during driver loading and reloading.
- **Driver dispatch table (DDT)**—Lists some of the driver's entry points to which the operating system transfers control. The function decision table lists other entry points.
- **Function decision table (FDT)**—Provides the names of action routines for I/O functions the driver supports and indicates which of those functions it supports by using an intermediate system buffer.

OpenVMS Alpha provides prototype tables in object form and uses run time initialization routines to override the default table values provided by the object module. The prototype tables are contained in the object module IOC$DRIVER_TABLES.OBJ, which resides in the VMS$VOLATILE_PRIVATE_INTERFACES.OLB object library in SYS$LIBRARY. This object library is included by the driver link procedure.

To override default values in the prototype tables, the driver writer must code a driver initialization routine (DRIVER$INIT_TABLES). This initialization routine is called by the driver loader with no explicit input parameters, and it returns an integer status code. Implicit input parameters are the prototype DDT, DPT, and FDT tables whose global names are `driver$ddt`, `driver$dpt`, and `driver$fdt` respectively.

Table initialization macros and functions for use in the DRIVER$INIT_TABLES routine are available in the `vms_drivers.h` header file. Each macro invokes the corresponding function, checks the status it returns and returns to the driver loader if it encounters an error.

The following sections in this chapter describe the DPT, DDT, and FDT tables and explain how a driver must finish setting up these tables in its own DRIVER$INIT_TABLES routine.

10.3 Driver Prologue Table

The driver prologue table (DPT), along with parameters to the System Management utility (SYSMAN) command that request driver loading, describes the driver to the driver-loading procedure. The driver-loading procedure performs the following tasks:

- Further initializes the DPT

- Creates data structures for the new devices in the I/O database
- Calls the OpenVMS executive loader to compute the size of the driver and load it into nonpaged system memory
- Links the new DPT into a list of all DPTs known to the system (IOC$GL_DPTLIST).

Device drivers can pass data-structure initialization information to the driver-loading procedure through values stored in the DPT. In addition, the driver-loading procedure initializes some fields within the I/O database using information from its own tables.

The macros listed in Table 10–1 should be used in the DRIVER$INIT_TABLES routine to finish the initialization of the prototype Driver Prologue Table. These macros invoke the appropriate initialization function and check the status returned from the function. If an error is returned, the macro returns control back to the driver loader with the error status.

The first parameter for all the DPT initialization macros is a pointer to the driver's DPT structure. Typically, this first parameter will be the address of the prototype DPT.

The values shown in the last column of the table are the initial values that are contained in the prototype DPT structure. These initial values can be changed by using the corresponding macro.

Table 10–1 DPT Initialization Macros for C

Macro name	DPT Field	Data type of second parameter	Prototype value
ini_dpt_adapt	dpt$il_adapt	integer	AT$_UBA
ini_dpt_bt_order	dpt$is_bt_order	integer	0
ini_dpt_decode	dpt$l_decw_sname	integer	0
ini_dpt_defunits	dpt$iw_defunits	integer	1
ini_dpt_deliver	dpt$ps_deliver	function pointer	0
ini_dpt_end[1]	-	-	-
ini_dpt_flags	dpt$il_flags	integer	DPT$M_ SMPMOD[2]
ini_dpt_idb_crams	dpt$iw_idbcrams idb_ crams	integer	0
ini_dpt_iohandles	dpt$il_loader_handle	integer	0
ini_dpt_maxunits	dpt$iw_maxunits	integer	8
ini_dpt_name	dpt$t_name	string pointer	-
ini_dpt_struc_init	dpt$ps_init_pd	function pointer	ioc$return
ini_dpt_struc_reinit	dpt$ps_reinit_pd	function pointer	ioc$return
ini_dpt_ucb_crams	dpt$iw_ucbcrams ucb_ crams	integer	0
ini_dpt_ucbsize	dpt$iw_ucbsize	integer	0
ini_dpt_unload	dpt$ps_unload	function pointer	0
ini_dpt_vector	dpt$ps_vector	pointer to vector of pointers	0

[1]The `ini_dpt_end` macro should be used immediately after all the other DPT initialization macros in the DRIVER$INIT_TABLES routine.

[2]The integer value specified with the `ini_dpt_flags` macro is logically-ORed with the default value and any previously specified values.

The following example shows the usage of the DPT initialization macros.

```
extern DPT driver$dpt;  /* Declare protype DPT */

ini_dpt_name        (&driver$dpt, "XXDRIVER");
ini_dpt_ucbsize     (&driver$dpt, sizeof(XX_UCB));
ini_dpt_adapt       (&driver$dpt, AT$_NULL);
ini_dpt_end         (&driver$dpt);
```

10.4 Driver Dispatch Table

The driver dispatch table (DDT) identifies those driver routines that the operating system calls to process I/O requests. Every driver must have a DDT.

The macros listed in Table 10–2 should be used in the DRIVER$INIT_TABLES routine to finish the initialization of the prototype Driver Dispatch Table.

The first parameter for all the DDT initialization macros is a pointer to the driver's DDT structure. Typically, this first parameter is the address of the prototype DDT.

The values shown in the last column of the table are the initial values that are contained in the prototype DDT structure.

The DDT initialization macros allow you to specify the names of the following routines, if applicable:

- Start-I/O routine
- Controller initialization routine
- Cancel-I/O routine
- Register dumping routine
- Unit initialization routine
- Alternate start-I/O routine
- Mount verification routine
- Cloned UCB routine
- Start-I/O routine for a driver employing the kernel process services

Table 10–2 DDT Macros

Macro name	DDT field	Data type of second parameter	Prototype value
ini_ddt_altstart	ddt$ps_altstart_2	function pointer	ioc$return_ success
ini_ddt_aux_routine	ddt$ps_aux_routine	function pointer	ioc$return
ini_ddt_aux_storage	ddt$ps_aux_storage	address pointer	ioc$return
ini_ddt_cancel	ddt$ps_cancel_2	function pointer	ioc$return_ success
ini_ddt_cancel_selective	ddt$ps_cancel_selective	function pointer	ioc$return_ unsupported
ini_ddt_channel_assign	ddt$ps_channel_assign_ 2	function pointer	ioc$return_ unsupported
ini_ddt_cloneducb	ddt$ps_cloneducb_2	function pointer	ioc$return_ success
ini_ddt_ctrlinit	ddt$ps_ctrlinit_2	function pointer	ioc$return_ success
ini_ddt_diagbf	ddt$iw_diagbuf	integer	0
ini_ddt_erlgbf	ddt$iw_errorbuf	integer	0
ini_ddt_kp_reg_mask	ddt$is_reg_mask	integer	0
ini_ddt_kp_stack_size	ddt$is_stack_bcnt	integer	0
ini_ddt_kp_startio	ddt$ps_kp_startio	function pointer	ioc$return
ini_ddt_mntv_for	ddt$ps_mntv_for	function pointer	ioc$return
ini_ddt_mntver	ddt$ps_mntver_2	function pointer	ioc_std$mntver
ini_ddt_regdmp	ddt$ps_regdump_2	function pointer	ioc$return_ success
ini_ddt_start	ddt$ps_start_2	function pointer	ioc$return_ success
ini_ddt_unitinit	ddt$ps_unitinit_2	function pointer	ioc$return_ success
ini_ddt_end[1]	-	-	-
ini_csr_mapping	ddt$ps_csr_mapping		ioc$return_ successs

[1]The `ini_ddt_end` macro should be used immediately after all the other DDT initialization macros in the DRIVER$INIT_TABLES routine.

10.5 Function Decision Table

The function decision table (FDT) is a structure within a driver that is used to select the appropriate device-dependent preprocessing routine for each one of the 64 possible I/O function codes. The FDT consists of two substructures:

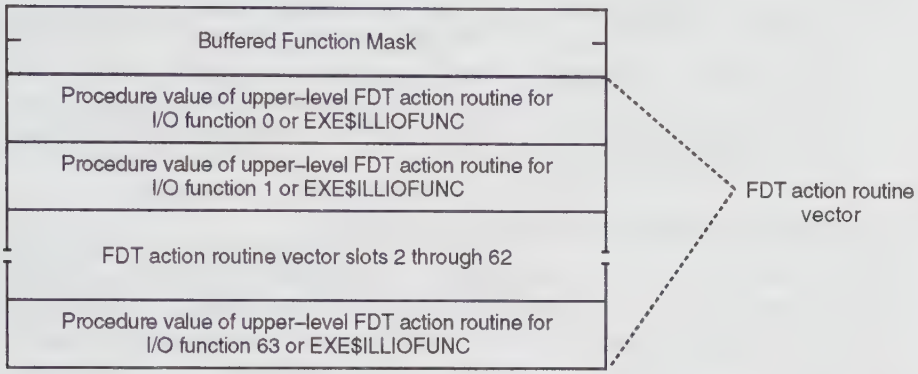
- A quadword bitmap known as the buffered function mask
- A 64-element array of longwords known as the FDT action routine vector

Each bit in the **buffered function mask** represents an I/O function that is serviced by the driver by means of an intermediate system buffer. (The distinctions between buffered and direct I/O functions will be discussed in Section 10.5.3.)

Each 32-bit slot of the **FDT action routine vector** corresponds to the symbolic value of an I/O function defined by the `iodef.h` header file in the SYS$STARLET_C.TLB text library. (Many of these function codes are listed in Table 10–3.) Those vector slots that relate to functions serviced by a driver contain the procedure value of an upper-level FDT action routine that initiates driver-specific preprocessing of the function. Those slots that represent functions the driver does not support contain the procedure value of a system upper-level FDT action routine that processes illegal I/O functions (EXE$ILLIOFUNC). When a $QIO is issued to a device driver, specifying an I/O function code the driver does not support, EXE$ILLIOFUNC executes, calling the FDT completion routine EXE_STD$ABORTIO. EXE_STD$ABORTIO terminates the I/O request and passes SS$_ILLIOFUNC status back to the $QIO caller.

Figure 10–1 depicts the layout of the FDT.

Figure 10–1 Layout of Function Decision Table (FDT)



The prototype FDT specifies the routine EXE$ILLIOFUNC as the upper-level FDT action routine for all functions and specifies that no functions are buffered.

The `ini_fdt_act` and `ini_fdt_end` macros should be used in the DRIVER$INIT_TABLES routine to finish the initialization of the prototype Function Decision Table. Each valid function code that can be processed must be associated with an upper-level FDT action routine by use of the `ini_fdt_act` macro.

```
ini_fdt_act (fdt, func, action, bufflag)
```

The parameters are:

fdt	Pointer to the FDT structure. Usually the address of the prototype FDT, `driver$fdt`
func	I/O function code.
action	Upper-level FDT action routine that is to be called from the specified function code.
bufflag	Specifies whether the function is a buffer or direct. The bufflag parameter has the value BUFFERED if the function is buffered, NOT_BUFFERED or DIRECT otherwise.

An upper-level FDT action routine must either be an FDT completion routine or must eventually transfer control to an FDT completion routine. An FDT completion routine either queues an I/O request packet (IRP) to a driver, inserts an IRP in the postprocessing queue, or aborts the I/O request. See Chapter 6 for additional information on upper-level FDT action routines, FDT support routines, and FDT completion routines.

The ini_fdt_end macro should be used immediately after all the other ini_fdt_act macros in the DRIVER$INIT_TABLES routine.

The following example shows the usage of the FDT initialization macros.

```
extern FDT driver$fdt;  /* Declare protype FDT */

ini_fdt_act ($driver$fdt, IO$_READLBLK, my_read_fdt, DIRECT);
ini_fdt_act ($driver$fdt, IO$_ACCESS,   my_access_fdt, BUFFERED);
ini_fdt_end (&driver$fdt);
```

10.5.1 OpenVMS Alpha I/O Function Codes

Table 10–3 lists the physical, logical, and virtual I/O function codes defined by the operating system. A complete list of function codes and values is contained in the iodef.h header file in the SYS$STARLET_C.TLB text library.

Table 10–3 I/O Function Codes

Function	Description	Equivalent Symbols
Physical I/O		
IO$_UNLOAD	Unload drive (required by all disk drivers)	IO$_LOADMCODE† (load microcode), IO$_START_BUS† (start LAVc bus)
IO$_SEEK	Seek cylinder	IO$_SPACEFILE† (space files), IO$_STARTMPROC† (start microprocessor), IO$_STOP_BUS† (stop LAVc bus)
IO$_PACKACK	Pack acknowledgment (required by all disk drivers)	IO$_STOP_MONITOR† (stop LAVc channel monitor)
IO$_SEARCH	Search for sector	IO$_SPACERECORD† (space records), IO$_READRCT† (read replacement and caching table)
IO$_WRITECHECK	Write check data	–
IO$_WRITEPBLK	Write physical block	–
IO$_READPBLK	Read physical block	–
IO$_AVAILABLE	Set device available (required by all disk drivers)	–
IO$_DSE	Data security erase (and rewind)	–
IO$_SETCHAR	Set device characteristics	–

†Unsupported; subject to change without notice.

(continued on next page)

Table 10–3 (Cont.) I/O Function Codes

Function	Description	Equivalent Symbols
Physical I/O		
IO$_SENSECHAR	Sense device characteristics	–
IO$_FORMAT	Format	IO$_CLEAN† (clean tape)
Logical I/O		
IO$_WRITELBLK	Write logical block	–
IO$_READLBLK	Read logical block	–
IO$_REWINDOFF	Rewind and set offline	IO$_READRCTL† (read RCT sector 0)
IO$_SETMODE	Set mode	–
IO$_REWIND	Rewind tape	–
IO$_SKIPFILE	Skip files	IO$_PSXSETMODE† (POSIX set mode)
IO$_SKIPRECORD	Skip records	IO$_PSXSENSEMODE† (POSIX sense mode)
IO$_SENSEMODE	Sense mode	–
IO$_WRITEOF	Write end of file	IO$_TTY_PORT_BUFIO† (Terminal port driver FDT routine for buffered I/O)
Virtual I/O		
IO$_WRITEVBLK	Write virtual block	–
IO$_READVBLK	Read virtual block	–
IO$_ACCESS	Access file	IO$_PSXWRITEVBLK† (POSIX write virtual block)
IO$_CREATE	Create file	–
IO$_DEACCESS	Deaccess file	IO$_PSXREADVBLK† (POSIX read virtual block)
IO$_DELETE	Delete file	–
IO$_MODIFY	Modify file	IO$_NETCONTROL† (X.25 network control function)

(continued on next page)

Table 10–3 (Cont.) I/O Function Codes

Function	Description	Equivalent Symbols
Virtual I/O		
IO$_READPROMPT	Read terminal with prompt	IO$_SETCLOCK (set clock), IO$_AUDIO (CD–ROM audio)
IO$_ACPCONTROL	Miscellaneous ACP control	IO$_STARTDATA (start data)
IO$_CONINTREAD	Connect to interrupt read-only	–
IO$_CONINTWRITE	Connect to interrupt with write	–

10.5.2 Defining Device-Specific Function Codes

You can also define device-specific function codes by equating the name of a device-specific function with the name of an existing function that is irrelevant to the device. The selected codes should, however, have a type (logical, physical, or virtual) that is appropriate for the function they represent. Also, user programs that issue $QIO requests specifying a device-specific code must similarly redefine the existing function. For example, the C code that follows defines three device-specific physical I/O function codes.

```
#define IO$_STARTCLOCK  IO$_ERASETAPE      /* Start interval clock */
#define IO$_STARTDATA   IO$_SPACEFILE      /* Start data acquisition */
```

10.5.3 Choosing Buffered I/O vs. Direct I/O

In selecting the functions that are to be buffered, consider the following:

- Direct I/O is intended only for devices whose I/O operations always complete quickly. For example, although terminal I/O appears fast, users can prevent the I/O operation from completing by using Ctrl/S to halt the operation indefinitely; therefore, terminal I/O operations are buffered I/O.
- Use of direct I/O requires that the process pages containing the buffer be locked in memory. Locking pages in memory increases the overhead of swapping the process that contains the pages.
- Use of buffered I/O requires that the data be moved from the system buffer to the user buffer. Moving data requires additional time.
- Routines that manipulate data before delivering it to the user (for example, an interrupt service routine for a terminal) cannot gain access to the data if direct I/O is used. Therefore, transfers that require data manipulation must be buffered I/O.
- The operating system handles the quotas differently for direct I/O and buffered I/O.

- Generally, direct-memory-access (DMA) devices use direct I/O, while programmed I/O devices use buffered I/O.

10.6 Device Database Initialization/Reinitialization

The initialization and reinitialization function addresses are stored in `dpt$ps_init_pd` and `dpt$ps_reinit_pd` respectively. The function calls are:

```
func (crb,ddb,idb,orb,ucb);
```

The parameters are:

crb	Channel Request Block address
ddb	Device Data Block address
idb	Interrupt Data Block address
orb	Owner Rights Block address
ucb	Unit Control Block address

The `dpt_store_isr` and `dpt_store_isr_vec` macros are used to store the procedure descriptor and entry point addresses of an interrupt service routine in a VEC entry in a given Channel Request Block. The `dpt_store_isr` macro fills in the first or only VEC entry in a CRB. The `dpt_store_isr_vec` macro allows the index of the VEC entry to be supplied. The formats of the macros are as follows:

```
dpt_store_isr (crb, isr);
dpt_store_isr_vec( crb, vecno, isr);
```

crb	Channel Request Block address.
vecno	Index (0, 1, 2...) of VEC entry to be filled in.
isr	Interrupt service routine address.

Part III

Running OpenVMS Alpha Device Drivers

Once you have written a device driver, you have to compile, link, and load it into the operating system. If you run into problems, you'll have to debug it, too!

Part III describes how to compile, link, load, and debug an OpenVMS Alpha device driver. It includes the following chapters:

- Chapter 11 describes how to compile and link a user-written OpenVMS Alpha driver.
- Chapter 12 explains how to load a user-written OpenVMS Alpha driver into the operating system.
- Chapter 13 describes how to debug a user-written OpenVMS Alpha driver.

11

Compiling and Linking a Device Driver

After your device driver is coded, it must be compiled into object modules and then linked to create one executive image. Once a device driver has been linked correctly, it is ready to be loaded into the operating system as described in Chapter 12.

This chapter describes the commands and qualifiers you use to compile and link an OpenVMS Alpha device driver.

11.1 Compiling a Driver

To compile an OpenVMS Alpha device driver, you must use the DEC C for OpenVMS Alpha compiler. DEC C is available as an optional Digital software development product.

This section contains a sample compile command, and it briefly describes the DEC C compiler options typically used to compile a device driver module. For additional information about these and other compiler qualifiers and parameters, consult your DEC C documentation.

To see an example of a command procedure that includes a representative command for use in compiling an OpenVMS Alpha device driver, refer to Appendix B. This sample file is also available in SYS$EXAMPLES:LRDRIVER.COM.

Use the following compile command line as the model for the command line to compile an OpenVMS Alpha device driver.

```
$ CC/STANDARD=RELAXED_ANSI89/INSTRUCTION=NOFLOATING_POINT-
          /EXTERN=STRICT-
          /POINTER_SIZE=32-
          /DEBUG-
         /LIS=LIS$xxDRIVER/MACHINE_CODE-
        /OBJ=OBJ$xxDRIVER-
               SRC$:xxDRIVER -
               +SYS$LIBRARY:SYS$LIB_C.TLB/LIBRARY
```

/STANDARD=RELAXED_ANSI
The DEC C compiler conforms to the ANSI C standard, but it allows various extensions required for compatibility with OpenVMS conventions. For example, in this mode the compiler accepts the dollar sign character in identifier names. This qualifier is required for compiling a driver module.

/INSTRUCTION=NOFLOATING_POINT
Suppresses the generation of floating point instructions for integer operations. Code running in system context cannot use the floating point registers. This qualifier is required for compiling a driver module.

/EXTERN_MODEL=STRICT_REFDEF
Causes the compiler to treat external references and definitions in the manner that is most compatible with the way that OpenVMS privileged global data cells are defined. This qualifier is required for compiling a driver module.

/POINTER_SIZE=32
Enables pointer-size features and sets the default pointer size to 32-bits long. This qualifier is required only if your driver includes support for 64-bit virtual addressing. Because the example device driver supports 64-bit virtual addresses for user buffers, it must be compiled with this qualifier.

/LIST=file_name
Specifies a compiler output listing file name. This qualifier is optional for compiling device drivers.

/MACHINE_CODE
The compiler output listing includes the generated machine code. This qualifier is optional for compiling device drivers.

/OBJECT=file_name
Specifies a compiler output object file name. This qualifier is optional for compiling device drivers.

/DEBUG
The compiler includes information in the object module that enables you to effectively use the System-Code Debugger on this module. This extra information is placed into the debug symbol file when the /DSF qualifier is specified when the driver image is linked. The /DEBUG qualifier does not affect the efficiency of the generated code and is optional for compiling device drivers.

+SYS$LIBRARY:SYS$LIB_C.TLB/LIBRARY
This command line element specifies a text library that is to be searched by the compiler when processing `#include` directives. This text library contains the header file modules that define various OpenVMS internal data structures and system macros. This library is required for compiling most driver modules.

11.2 Linking a Driver

This section contains a sample of the LINK command for linking an OpenVMS Alpha device driver, and it briefly describes LINK command qualifiers used to link device drivers.

To see an example of a command procedure that includes a representative command and options file for use in linking an OpenVMS Alpha device driver, refer to Appendix B. This sample file is also available in SYS$EXAMPLES:LRDRIVER.COM.

Use the following LINK command line as the model for the command line to link an OpenVMS Alpha device driver.

```
$ LINK/ALPHA/NATIVE_ONLY/BPAGE=14-
            /SECTION_BINDING/NOTRACEBACK -
            /NODEMAND_ZERO/SHAREABLE=xxDRIVER -
            /SYSEXE=SELECTIVE/NOSYSSHR/DSF=xxDRIVER -
            /MAP=xxDRIVER /FULL /CROSS_REFERENCE -
            xxDRIVER_LNK/OPTION
```

The qualifiers used in this command line are as follows:.

/ALPHA
Directs the linker to create an Alpha image. This is the default qualifier on Alpha systems, and it is required for linking OpenVMS Alpha device drivers.

/NATIVE_ONLY
Indicates that there will be no calls to translated shareable images from this image. This is the default qualifier on Alpha systems, and it is required for linking OpenVMS Alpha device drivers.

/BPAGE=14
Specifies the page size the linker should use when it creates the image sections that make up the image. The value **14** is usually specified for executive images and indicates that the linker should lay out image sections on 16KB boundaries. This qualifier is optional for linking device drivers.

The driver-loading procedure ignores the image section boundaries defined by the linker if the image is being loaded as a sliced executive image.

/SECTION_BINDING
Directs the linker to activate section binding for both code and data image sections in the driver. Upon successful binding of code sections and data sections, the linker sets bits EIHD$V_BIND_CODE and EIHD$V_BIND_DATA in the image's header. If either of these bits is not set, the driver-loading procedure does not load the driver image as a sliced executive image, but rather, performs a normal load of the image. This qualifier is optional for linking device drivers.

/NOTRACEBACK
Directs the linker to omit traceback information from the image. This qualifier is required for linking device drivers.

/NODEMAND_ZERO
Directs the linker to inhibit generation of demand-zero sections in a driver executive image. This qualifier is required for linking device drivers.

/SHAREABLE=*xx*DRIVER
Directs the linker to create a shareable executive image named *xx*DRIVER.EXE. This qualifier is required for linking device drivers.

/SYSEXE
Directs the linker to selectively search the system shareable image, SYS$BASE_IMAGE.EXE, to resolve symbols in a link operation. When the linker selectively searches SYS$BASE_IMAGE.EXE, it only includes symbols from the SYS$BASE_IMAGE.EXE global symbol table that were referenced by input files previously processed in the link operation. This qualifier is required for linking device drivers.

/DSF
Directs the linker to create a file called a debug symbol file (DSF) for use by the OpenVMS Alpha System-Code Debugger. Specify the character string you want the linker to use as the name of for the debug symbol file. If you do not include a file type in the character string, the linker appends the .DSF file type to the file name. This qualifier is optional for linking device drivers.

/NOSSYSHR
Directs the linker not to search the system default shareable image library (SYS$LIBRARY:IMAGELIB.OLB) to resolve symbolic references. Drivers may not resolve symbols from shareable image libraries. This qualifier is required for linking device drivers.

/MAP=*xx*DRIVER
Directs the linker to create an image map file. You need a map file for the driver image to assist in debugging. This qualifier is optional for linking device drivers.

/FULL
Directs the linker to create a full image map. This qualifier is optional for linking device drivers.

/CROSS_REFERENCE
Directs the linker to place the Symbols by Name section in the image map with the Symbols Cross-reference section. This qualifier is optional for linking device drivers.

***xx*DRIVER_LNK/OPTION**
Identifies the input file specification (here, *xx*DRIVER_LNK) as a linker options file.

In the linker options file, you must specify the object modules compiled for your driver and other linker options required to build an executive image. This qualifier is required for linking device drivers.

The options file used to link the example device driver is included in the LRDRIVER.COM command procedure in Appendix B.

12

Loading a Device Driver

An OpenVMS Alpha device driver is created as an executive image, and it is loaded as an integral part of the executive by the executive loader. You can load a non-Digital-supplied device driver any time after the system is booted.

This chapter describes the following methods for loading OpenVMS Alpha device drivers into the OpenVMS Alpha operating system:

- Using the System Management utility (SYSMAN).
- Writing your own IOGEN Configuration Building Module (ICBM).

In addition to these sections, Section 12.2 explains how to enhance system performance by loading as "sliced" images drivers that have been linked with section binding enabled.

12.1 Using SYSMAN to Configure and Load Drivers

You can use SYSMAN to connect devices, load OpenVMS Alpha device drivers, and display configuration information useful for debugging device drivers.

To invoke SYSMAN, enter the following command:

```
$ MCR SYSMAN
```

The SYSMAN prompt `SYSMAN>` is displayed:

All SYSMAN commands that control and display the I/O configuration of an OpenVMS Alpha system must be introduced with the prefix IO. For example, to autoconfigure a system, enter the following commands:

```
$ MCR SYSMAN
SYSMAN> IO AUTOCONFIGURE
```

For adapters supported by Digital, there is never any need to connect a device manually. Use the SYSMAN IO AUTOCONFIGURE command with the appropriate /SELECT and /EXCLUDE lists to configure the system. If you omit these qualifiers, the IO AUTOCONFIGURE command configures the entire system.

For non-Digital-supplied adapters and new Digital adapters not yet supported by the IO AUTOCONFIGURE command, you must perform a manual connect, generally issuing an IO CONNECT command in the following format:

```
SYSMAN> IO CONNECT devname/ADAPTER=x/CSR=y/VECTOR=z/DRIVER=xxdriver-
                /node=busspecificinfo
```

In such a command, specifying the device name and driver name is straightforward (and described in Section 12.1). For more information about how to determine the `adapter`, `csr`, `vector`, and `node` parameters for devices attached to PCI, ISA, and EISA buses, see the appropriate bus support chapter in this manual.

The following sections describe the SYSMAN commands you can use to load an OpenVMS Alpha device driver.

AUTOCONFIGURE

Automatically identifies and configures all hardware devices attached to a system. The AUTOCONFIGURE command connects devices and loads their drivers.

You must have CMKRNL and SYSLCK privileges to use the AUTOCONFIGURE command.

Format

IO AUTOCONFIGURE

Parameters

None.

Description

The AUTOCONFIGURE command identifies and configures all hardware devices attached to a system. It connects devices and loads their drivers. You must have CMKRNL and SYSLCK privileges to use the AUTOCONFIGURE command.

Qualifiers

/SELECT=(device_name[,...])
Specifies the device type to be automatically configured. Use valid device names or mnemonics that indicate the devices to be included in the configuration. Wildcards must be explicitly specified.

The /SELECT and /EXCLUDE qualifiers are not mutually exclusive, as they are on OpenVMS VAX. Both qualifiers can be specified on the command line.

Table 12–1 shows /SELECT qualifier examples.

Table 12–1 SELECT Qualifier Examples

Command	Devices that are configured	Devices that are not configured
/SELECT=P*	PKA,PKB,PIA	None

(continued on next page)

Table 12–1 (Cont.) SELECT Qualifier Examples

Command	Devices that are configured	Devices that are not configured
/SELECT=PK*	PKA,PKB	PIA
/SELECT=PKA*	PKA	PKB,PIA

/EXCLUDE=(device_name[,...])
Specifies the device type that should not be automatically configured. Use valid device names or mnemonics that indicate the devices to be excluded from the configuration. Wildcards must be explicitly specified.

The /SELECT and /EXCLUDE qualifiers are not mutually exclusive, as they are on OpenVMS VAX systems. Both qualifiers can be specified on the command line.

/LOG
Controls whether the AUTOCONFIGURE command displays information about loaded devices.

CONNECT

Connects a hardware device and loads its driver, if the driver is not already loaded.

You must have CMKRNL and SYSLCK privileges to use the CONNECT command.

Format

IO CONNECT device-name[:]

Parameters

device-name[:]
Specifies the name of the hardware device to be connected. It should be specified in the format device-type, controller, and unit number (for example, LPA0 where LP is a line printer on controller A at unit number 0). If the /NOADAPTER qualifier is specified, the device is the software device to be loaded.

Description

The CONNECT command connects a hardware device and loads its driver, if the driver is not already loaded. You must have CMKRNL and SYSLCK privileges to use the CONNECT command.

Qualifiers

/ADAPTER=tr_number
/NOADAPTER (default)
Specifies the nexus number of the adapter to which the specified device is connected. It is a nonnegative 32-bit integer. /NOADAPTER indicates that the device is not associated with any particular hardware. The /NOADAPTER qualifier is compatible with the /DRIVER_NAME qualifier only.

/CSR=csr_address
The CSR address for the device being configured. This address must be specified in hexadecimal. You must precede the CSR address with %X. The CSR address is a quadword value that is loaded into IDB$Q_CSR without any interpretation by SYSMAN. This address can be physical or virtual depending on the specific device being connected:

- /CSR=%X3A0140120 for a physical address

- /CSR=%XFFFFFFFF807F8000 for a virtual address (the sign extension is required for OpenVMS Alpha virtual addresses)

This qualifier is required if /ADAPTER=tr_number is specified.

/DRIVER_NAME=filespec
The name of the device driver to be loaded. If this qualifier is not specified, the default is obtained in the same manner as the SYSGEN default name. For example, if you want to load the Digital-supplied SYS$ELDRIVER.EXE, the "SYS$" must be present. Without the "SYS$", SYSMAN looks for ELDRIVER.EXE in SYS$LOADABLE_IMAGES. This implementation separates the user device driver namespace from Digital-supplied device driver namespace.

/LOG=(ALL,CRB,DDB,DPT,IDB,SB,UCB)
/NOLOG (default)
Controls whether SYSMAN displays the addresses of the specified control blocks. The default value for the /LOG qualifier is /LOG=ALL. If /LOG=UCB is specified, a message similar to the following is displayed:

```
%SYSMAN-I-IOADDRESS, the UCB is located at address 805AB000
```

The default is /NOLOG.

/MAX_UNITS=maximum-number-of-units
Specifies the maximum number of units the driver can support. The default is specified in the Driver Prologue Table (DPT) of the driver. If the number is not specified in the DPT, the default is 8. This number must be greater than or equal to the number of units specified by /NUM_UNITS. This qualifier is optional.

/NUM_UNITS=number-of-units
Specifies the number of units to be created. The starting device number is the number specified in the device name parameter. For example, the first device in DKA0 is 0. Subsequent devices are numbered sequentially. The default is 1. This qualifier is optional.

/NUM_VEC=vector-count
Specifies the number of vectors for this device. The default vector count is 1. The /NUM_VEC qualifier is optional. This qualifier should be used only when using the /VECTOR_SPACING qualifier. When using the /NUM_VEC qualifier, you must also use the /VECTOR qualifier to supply the base vector.

/SYS_ID=number-of-remote-system
Indicates the SCS system ID of the remote system to which the device is to be connected. It is a 64-bit integer; you must specify the remote system number in hexadecimal. The default is the local system. This qualifier is optional.

/VECTOR=(vector-address,...)
The interrupt vectors for the device or lowest vector. This is a byte offset into the SCB of the interrupt vector for directly vectored interrupts or a byte offset into the ADP vector table for indirectly vectored interrupts. The values must be longword aligned. To specify the vector address(es) in octal or hexadecimal, precede the address(es) with %O or %X, respectively. This qualifier is required when /ADAPTER=tr_number or /NUM_VEC=vector-count is specified. Up to 64 vectors can be listed.

/VECTOR_SPACING=number-of-bytes-between-vectors
Specifies the spacing between vectors. Specify the amount as a multiple of 16 bytes. The default is 16. You must specify both the base vector with /VECTOR and the number of vectors with /NUM_VEC. This qualifier is optional.

Examples

1.
```
SYSMAN> IO CONNECT DKA0:/DRIVER_NAME=SYS$DKDRIVER/CSR=%X80AD00-
/ADAPTER=4/NUM_VEC=3/VECTOR_SPACING=%X10/VECTOR=%XA20/LOG
%SYSMAN-I-IOADDRESS, the CRB is located at address 805AEC40
%SYSMAN-I-IOADDRESS, the DDB is located at address 805AA740
%SYSMAN-I-IOADDRESS, the DPT is located at address 80D2A000
%SYSMAN-I-IOADDRESS, the IDB is located at address 805AEE80
%SYSMAN-I-IOADDRESS, the SB is located at address 80417F80
%SYSMAN-I-IOADDRESS, the UCB is located at address 805B68C0
```

2.
```
SYSMAN> IO CONNECT DKA0:/DRIVER_NAME=SYS$DKDRIVER/CSR=%X80AD00-
/ADAPTER=4/VECTOR=(%XA20,%XA30,%XA40)/LOG=(CRB,DPT,UCB)
%SYSMAN-I-IOADDRESS, the CRB is located at address 805AEC40
%SYSMAN-I-IOADDRESS, the DPT is located at address 80D2A000
%SYSMAN-I-IOADDRESS, the UCB is located at address 805B68C0
```

3.
```
SYSMAN> IO CONNECT FTA0:/DRIVER=SYS$FTDRIVER/NOADAPTER/LOG=(ALL)
%SYSMAN-I-IOADDRESS, the CRB is located at address 805AEC40
%SYSMAN-I-IOADDRESS, the DDB is located at address 805AA740
%SYSMAN-I-IOADDRESS, the DPT is located at address 80D2A000
%SYSMAN-I-IOADDRESS, the IDB is located at address 805AEE80
%SYSMAN-I-IOADDRESS, the SB is located at address 80417F80
%SYSMAN-I-IOADDRESS, the UCB is located at address 805B68C0
```

4.
```
SYSMAN> IO CONNECT FTA1:/DRIVER=SYS$FTDRIVER/NOADAPTER
SYSMAN>
```

SET PREFIX

Sets the prefix list which is used to manufacture the IOGEN Configuration Building Module (ICBM) names.

Format

IO SET PREFIX=(icbm_prefix[,...])

Parameters

icbm_prefix[,...]
Specifies ICBM prefixes. These prefixes are used by the AUTOCONFIGURE command to build ICBM image names.

Description

The SET PREFIX command sets the prefix list which is used to manufacture SYSMAN Configuration Building Module (ICBM) names.

Qualifiers

None.

Example

```
SYSMAN> IO SET PREFIX=(SYS$,PSI$,VME_)
```

SHOW DEVICE

Displays information on device drivers loaded into the system, the devices connected to them, and their I/O databases. All addresses are in hexadecimal and are virtual.

Format

IO SHOW DEVICE

Parameters

None.

Qualifiers

None.

Description

The SHOW DEVICE command displays information on the device drivers loaded into the system, the devices connected to them, and their I/O databases.

The SHOW DEVICE command specifies that the following information be displayed about the specified device driver:

Driver	Name of the driver
Dev	Name of each device connected to the driver
DDB	Address of the device's device data block
CRB	Address of the device's channel request block
IDB	Address of the device's interrupt dispatch block
Unit	Number of each unit on the device
UCB	Address of each unit's unit control block

All addresses are in hexadecimal and are virtual.

Example

```
SYSMAN> IO SHOW DEVICE
```

The following is a sample display produced by the SYSMAN IO SHOW DEVICE command:

```
__Driver_______Dev_DDB_____CRB_____IDB_____Unit_UCB____
SYS$FTDRIVER
               FTA 802CE930 802D1250 802D04C0
                                                0 801C3710
SYS$EUDRIVER
               EUA 802D0D80 802D1330 802D0D10
                                                0 801E35A0
SYS$DKDRIVER
               DKI 802D0FB0 802D0F40 802D0E60
                                                0 801E2520
SYS$PKADRIVER
               PKI 802D1100 802D13A0 802D1090
                                                0 801E1210
SYS$TTDRIVER
OPERATOR
NLDRIVER
```

SHOW PREFIX

Displays the current prefix list used in the manufacture of ICBM names.

Format

IO SHOW PREFIX

Parameters

None.

Description

The SHOW PREFIX command displays the current prefix list on the console. This list is used by the AUTOCONFIGURE command to build ICBM names.

Qualifiers

None.

Example

```
SYSMAN> IO SHOW PREFIX
%SYSMAN-I-IOPREFIX, the current prefix list is: SYS$,PSI$,VME_
```

12.2 Loading Sliced Executive Images

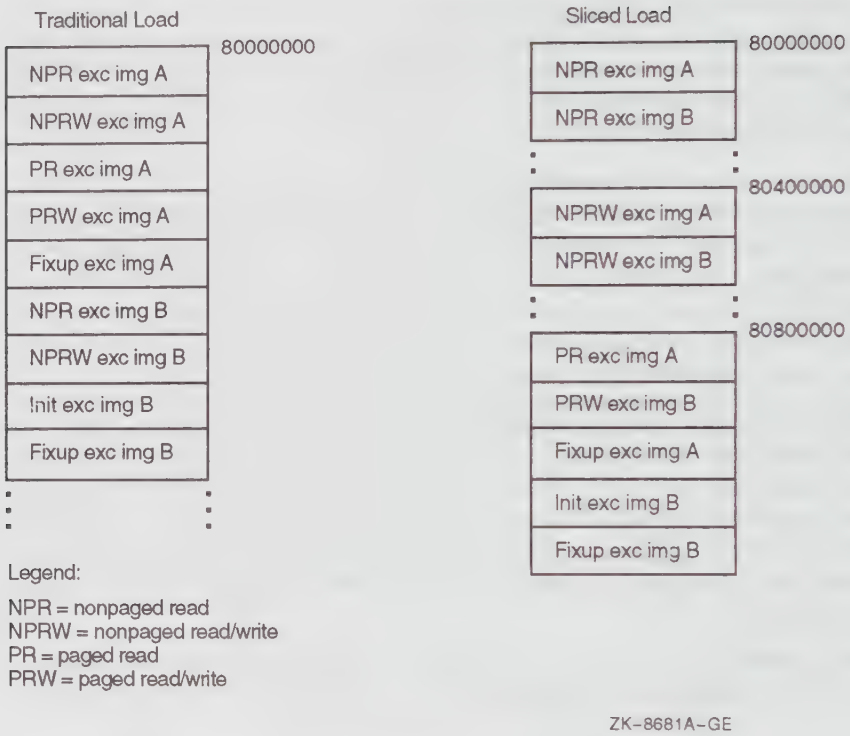
In traditional executive image loading, code and data are sparsely laid out in system address space. The loader allocates the virtual address space for executive images so that the image sections are loaded on the same boundaries as the linker created them. The images are normally linked with the /BPAGE qualifier equal to 14; this puts the image sections on 16 KB boundaries.

Alpha hardware can consider a set of pages as a single huge page, which can be mapped by a single page-table entry (PTE) in the translation buffer. To use this mechanism, the loader allocates a PTE for nonpaged code and another for nonpaged data. Pages within this huge page, or **granularity hint region**, must have the same protection. As a result, code and data cannot share a huge page. The end result of this is a single translation buffer entry to map the executive nonpaged code, and another to map the nonpaged data.

The loader then loads like nonpaged sections from each executive image into the same region of virtual memory, ignoring the page size according to which the image sections have been created. Paged, fixup and initialization sections are loaded in the same manner as the traditional loader. If the parameter S0_PAGING is set to turn off paging of the executive, all code and data, both paged and nonpaged, is treated as nonpaged and loaded into the granularity hint regions.

This method of loading is called "sliced" loading. Figure 12–1 illustrates a traditional load and a sliced load.

Figure 12–1 Traditional and Sliced Loads



12.2.1 Controlling Executive Image Slicing

The system parameter LOAD_SYS_IMAGES is a bitmask and has several bits defined:

SGN$V_LOAD_SYS_IMAGES (bit 0)	Enables loading alternate executive images specified in VMS$SYSTEM_IMAGES.DATA.
SGN$V_EXEC_SLICING (bit 1)	Enables the loading of the executive into granularity hint regions.
SGN$V_RELEASE_PFNS (bit 2)	Enables releasing unused portions of the huge pages.

These bits are set by default. Use conversational bootstrap to disable executive image slicing.

12.3 Writing an IOGEN Configuration Building Module (ICBM)

An **IOGEN Configuration Building Module (ICBM)** is a shareable image used to configure I/O devices on a system. Digital ships several ICBMs, which are used to configure the devices supported by the OpenVMS Alpha operating system. (For example, SYS$SHARE:SYS$ICBM_07.EXE loads the devices supported by the DEC 3000-300L system.)

The following sections explain ICBM processing and describe how to write an ICBM to load your user-written driver.

12.3.1 Quick Overview of ICBM Processing

When a user enters the SYSMAN IO AUTOCONFIGURE command, the Autoconfigure Utility (Autoconfigure) finds the ICBMs for specific platforms and activates them. Autoconfigure first calls the ICBM's initialization routine, which returns to Autoconfigure a table that lists the adapters known to the ICBM.

Autoconfigure then walks down the system ADP list. For each ADP, each ICBM that indicated it can process that ADP is called. The ICBM routine called scans the bus array structure pointed to by the ADP for devices it recognizes. For each device recognized, the ICBM loads the associated device driver and connects the appropriate number of units.

12.3.2 ICBM Structure

An ICBM consists of the following three major components:

- An **autoconfiguration bus mappping table (ABM)**, which lists the adapter types supported by the ICBM and the ICBM routine that processes each type of adapter.
- An initialization routine that is responsible for passing the address of the ABM to Autoconfigure.
- The routines that process each adapter, looking for devices and loading drivers when recognized devices are found.

The shareable image SYS$SHARE:IOGEN$SHARE.EXE provides services, which are called by the ICBM configuration routines, to do the work of getting a driver loaded and units connected. These services load drivers, implement logging (SYSMAN IO AUTO/LOG), and process the IO AUTO /SELECT and /EXCLUDE qualifiers.

The examples shown in this section are extracted from MMOV$ICBM, which is the OpenVMS Alpha ICBM that loads the Multimedia Services video and audio drivers.

12.3.2.1 Autoconfigure Bus Mapping Table

The autoconfigure bus mapping table (ABM) consists of a series of longword pairs, terminated by a pair of zeros. The first longword is the adatper bus type; the second longword is the ICBM routine that configures devices on that bus. For example, this ABM indicates to Autoconfigure that it can configure devices on a PCI, TURBOchannel, and ISA buses:

```
static int mmov_abm[] =
    {
    AT$_PCI, (int) mmov$configure_bus,
    AT$_TC,  (int) mmov$configure_bus,
    AT$_ISA, (int) mmov$configure_bus,
    0, 0
    };
```

The routine mmov$configure_bus is the ICBM configuration routine that locates devices and loads drivers for them.

A single ICBM can configure devices on multiple bus types, simply by including them in the ABM table, and by providing the appropriate routines. In fact, the same ICBM can be used to configure devices on different buses on different systems. (More about this later).

12.3.2.2 ICBM Initialization Routine

Each ICBM must include an intitialization routine named IOGEN$ICBM_INIT, which is called by Autoconfigure when the ICBM image is activated. This routine returns the address of the ABM as shown in the following example:

```
int iogen$icbm_init(int *abm)
    {
    *abm = (int) mmov_abm;
    return (int) &IOGEN$_ICBM_OK;
    }
```

IOGEN$ICBM_INIT returns the status code IOGEN$_ICBM_OK to indicate to Autoconfigure that a valid, properly built ICBM has been loaded. (The value IOGEN$_ICBM_OK has been exported by the IOGEN$SHARE shareable image, which is why the &IOGEN$_ICBM_OK construct is used.) If this routine returns anything else, Autoconfigure assumes that the ICBM is built incorrectly, or that some error occurred while activating it, and the ICBM will be ignored.

12.3.2.3 ICBM Configuration Routine

The ICBM configuration routines are called by Autoconfigure as it processes the ADP tree built at the time the system buses were probed. For each ADP in the tree, Autoconfigure scans the list of ABMs obtained from the activated ICBMs. Each time Autoconfigure finds a bus type that matches the ADP bus type, the associated configuration routine is called, using the following prototype:

```
int mmov$configure_bus (int handle, ADP *adp)
```

where:

`handle` is a 32-bit quantity that provides context for Autoconfigure utility routines.
`adp` is a pointer to the ADP being processed.

The configuration routine follows these steps:

1. Locates the bus array that describes the devices found during bus probing. This pointed to by `adp$ps_bus_array`.
2. For each device, compares the device ID against the list of devices the ICBM supports. The device ID is located in `busarray$q_hw_id`. The values in this field vary according to bus type as follows:

TURBOchannel	8-byte option identification string located at the start of the option ROM.
PCI	32-bit PCI ID field located at the first longword in PCI configuration space.
ISA	Value (up to 8 bytes) that is in the `handle` field as set up by the console's ISACFG command. (For more information about the ISACFG command, see Chapter 15.)

3. For each match, constructs a device name. The controller letter comes from the `busarray$b_ctrlltr` field. If this field is zero, then IOGEN$ASSIGN_CONTROLLER is called to obtain the controller letter.

   For more information about this step and those that follow, see the CONNECT_THE_DRIVER routine in the Appendix C.
4. For the newly constructed device name, calls IOGEN$AC_SELECT to check the device against the /SELECT and /EXCLUDE lists, which may have been specified on the SYSMAN IO AUTOCONFIGURE command. If this routine does not return SS$_NORMAL, exits.
5. Checks if the device has already been configured, by checking the busarray$v_no_reconnect bit in the `busarray$l_flags` field. If the device has already been configured, exits.

6. If the device is selected or not excluded, constructs an item list for the SYS$LOAD_DRIVER service. If loading a driver and connecting a single unit, the following qualifiers are required: adapter TR number, base CSR physical address, interrupt vector offset, node number, and address to receive pointer to the newly created CRB. All of these qualifiers, except the address to receive the pointer to the newly created CRB correspond to the qualifiers on the SYSMAN IO CONNECT command:

   ```
   SYSMAN> IO CONNECT dev /ADAPTER=x/NODE=x/CSR=x/VECTOR=x/DRIVER=xx
   ```

   The device and driver names are passed to SYS$LOAD_DRIVER as parameters. The values for these items are generally obtained from various fields in the bus array. For more information, see the description of the SYS$LOAD_DRIVER routine in Section 12.3.6.

7. If the driver load was successful, calls IOGEN$LOG to log the fact that a new device was connected.

8. Finally, writes the CRB address back into the bus array, and sets the `busarray$v_no_reconnect` bit in the `busarray$l_flags` field. These tasks are performed by a kernel mode routine in the ICBM. This work is performed in kernel mode because the ADP data structure is not writable from the ICBM context. (See the WRITE_CRB_RECONNECT routine in the example.)

12.3.3 Building an ICBM

To compile an ICBM source module, use the SYS$LIBRARY:SYS$LIB_C.TLB system macro library.

An ICBM image must be linked against the SYS$SHARE:IOGEN$SHARE.EXE shareable image, and it must include the IOGEN$ICBM_CONTROL routine from SYS$LIBRARY:STARLET.OLB. In addition, the /NOTRACEBACK and /SHARE linker qualifiers must be specified.

The following commands will compile (using the DECC compiler) and link the example ICBM:

```
$ CC /OBJECT/LIST/MACHINE/STANDARD=RELAXED_ANSI/INSTRUCTION=(NO_FLOAT) -
        MMOV$ICBM.C
$ LINK/ALPHA/BPAGE=14/NOTRACEBACK/SHARE=MMOV$ICBM.EXE/MAP/FULL/CROSS -
        SYS$INPUT.OPT/OPTIONS
MMOV$ICBM.OBJ,-
SYS$LIBRARY:STARLET.OLB/INCLUDE=(IOGEN$ICBM_CONTROL),-
SYS$SHARE:IOGEN$SHARE/SHARE
```

12.3.4 Loading an ICBM

To use an ICBM image, the following conventions must be followed:

1. The image must be placed in SYS$SHARE—this is where Autoconfigure looks for it.

2. The ICBM image name must include the system type and can optionally include the CPU type: MMOV$ICBM_xxyy.EXE, where xx is the low-byte of the system type, and yy is the low-byte of the CPU type. These values are defined in the header file HWRPBDEF.H, and can be obtained using these lexical functions in DCL:

   ```
   SYSTYPE = F$GETSYI("SYSTYPE")
   CPUTYPE = F$GETSYI("CPUTYPE")
   ```

   These values must be in hexadecimal, and there must be two digits. For example, on a DEC 3000-300L processor, the example ICBM can be named

   ```
   MMOV$ICBM_07.EXE
   MMOV$ICMB_0702.EXE
   ```

   Note that the same ICBM image can be used on multiple systems, as long as the image name is changed appropriately. The example ICBM, for instance, supports TURBOchannel and PCI systems. On a DEC 3000-300L TURBOchannel system, the image is named MMOV$ICBM_07.EXE; on an Alphastation 400/166 PCI system, the image is named MMOV$ICBM_0D.EXE. Both of these are the identical image, simply renamed as appropriate based on the system they are installed on.

3. Because Autoconfigure executes as a privileged image, any shareable image Autoconfigure activates must be installed as a known image, including any ICBMs. Also, if Autoconfigure is to execute as part of the system startup process, the ICBM image must be installed prior to the Autoconfigure step in startup. To accomplish this, the recommended method is to place commands similar to the following in SYS$MANAGER:SYCONFIG.COM:

   ```
   $ IF (.NOT. F$FILE_ATTRIBUTES("SYS$SHARE:MMOV$ICBM_07.EXE","KNOWN"))
   $ THEN
   $    INSTALL :== $INSTALL/COMMAND
   $    INSTALL ADD SYS$SHARE:MMOV$ICBM_07.EXE
   $ ENDIF
   ```

   This sequence is recommended because AUTOGEN executes SYCONFIG.COM before Autoconfigure during system startup.

4. Finally, Autoconfigure must explicitly be told to look for the ICBM image. This is accomplished using SYSMAN's prefix list. The prefix is the part of the image name that identifies the product (e.g., MMOV$ in the example we've been using). Digital recommends that you register the prefix in order to avoid conflicts with Digital and other vendors' products.

   To display the prefix list, enter the following command:

   ```
   SYSMAN> IO SHOW PREFIX
   ```

   This will display the current set of prefixes, or an empty string. The empty string equates to the prefix "SYS$", specifying the ICBMs shipped with OpenVMS.

   To set the prefix list, enter the following command:

   ```
   SYSMAN> IO SET PREFIX="SYS$,MMOV$"
   ```

   The SYS$ prefix must be included; otherwise the system ICBMs will not be executed, and the system will have no I/O devices. The prefix list is stored in the file SYS$MANAGER:IOGEN$PREFIX.DAT. IOGEN will create this file if it does not exist when you set a prefix list.

12.3.5 Debugging an ICBM

Debugging ICBMs is difficult. For one thing, they execute in Executive mode, so the only debugger you can use is DELTA, which requires that you have machine code listings and linker maps. Access to the OpenVMS Listings CD is a requirement, because you will need access to [IOGEN.LIS]AUTOCONFIGURE.LIS—this module contains the routines that call your ICBM, and most of the routines that provide services to your ICBM.

One way to debug your ICBM is to create a small main program that calls IOGEN$AUTOCONFIGURE directly. This gives you a way to invoke the debugger. Here's an example, called AUTO.C:

```
#include <descrip.h>

int iogen$autoconfigure(int, void *, int, int, int, int);

    main()
        {
        int status;
        static $DESCRIPTOR(prefixes,"MMOV$");

        status = iogen$autoconfigure(0,&prefixes,0,0,0,0);
        }
```

Compile and link this program /DEBUG, and link it against the SYS$SHARE:IOGEN$SHARE shareable image. Define this logical name to specify the DELTA debugger:

```
DEFINE LIB$DEBUG DELTA
```

When this image is invoked, the DELTA debugger gains control. Using this debugger, the listings of AUTO, your ICBM, and Autoconfigure, you can debug your ICBM.

Tips for Debugging ICBMs

1. You cannot set breakpoints or examine variables in your ICBM image until Autoconfigure has activated it. The simplest way to get started is to locate the call to your ICBM's IOGEN$ICBM_INIT routine in the routine activate_icbms in AUTOCONFIGURE.LIS. Set a breakpoint on this call, and execute to there. Then step into the routine, and Delta will display the routine's address. From there, using your ICBM's linker map, you can figure out other addresses inside your ICBM.

   As long as you are using the example main program shown above, your ICBM will be loaded at the same address every time. So, you do not have to go through the exercise of determining where your ICBM is loaded each time you run the image.

2. After you have debugged the IOGEN$ICBM_INIT routine, you can set a breakpoint at the point where your ICBM's configuration routine is called, in routine process_adp in AUTOCONFIGURE.LIS. Autoconfigure calls process_adp for every ADP in the system. This routine compares the `adp$l_adp_type` field against your ICBM's ABM, and when there's a match, it calls the corresponding routine in your ICBM. When you hit this breakpoint, your ICBM configuration routine is about to be called, and you can single-step into this routine.

3. It can be difficult to figure out what the C compiler is doing. You can run the System Dump Analyzer on the system while you are debugging your ICBM, and you can use SDA to locate system data structure addresses. This is sometimes easier than trying to find which register the compiler is using for these addresses.

4. Keep your configuration routine as simple as possible. The more processing this routine does, and the more branching, the more difficult it is to follow the machine code, especially while single-stepping.

12.3.6 ICBM IOGEN routines

This section describes the IOGEN routines used in ICBMs.

IOGEN$AC_SELECT

Processes the /SELECT and /EXCLUDE lists for the IO AUTOCONFIGURE command.

Prototype

```
int iogen$ac_select ( int handle, void * device)
```

Parameters

Name	Access	Description
handle	Input	32-bit Autoconfiguration context. This value was passed into the ICBM configuration routine.
device	Input	Pointer to a device name character string descriptor. This descriptor contains the device name to be checked against the /SELECT and /EXCLUDE qualifiers.

Return Values

SS$_NORMAL	device may be included
IOGEN$_EXCLUDE	device is to be excluded

Context

Executive mode. Called by an ICBM configuration routine to determine if the device should be configured.

Description

This routine checks the device name described by the device parameter against the list of devices specified by the /SELECT and /EXCLUDE qualifiers on the SYSMAN IO AUTOCONFIGURE command. If the select list is empty, all devices are implicitly included. If this list is not empty, all devices are implicitly excluded. After determining if a device is selected, either implicitly or explicitly, a check is made to see if it is explicitly excluded.

IOGEN$ASSIGN_CONTROLLER

Assigns a device controller letter.

Prototype

int iogen$assign_controller (int handle, char * device, BUSARRAYENTRY * ba)

Parameters

Name	Access	Description
handle	Input	32-bit Autoconfiguration context. This value was passed into the ICBM configuration routine.
device	Input	Pointer to the first to characters of the device name.
ba	Input	Pointer to the bus array entry for the device being configured. The controller letter assigned is returned in the busarray$b_ctrlltr field of the bus array entry.

Return Values

SS$_NORMAL	Controller letter successfully assigned.
SS$_NOPRIV	Process does not have CMKRNL privilege.
SS$_ABORT	No more controller letters—the device cannot be configured.
SS$_ACCVIO	Ran out of workspace.

Context

Executive Mode. Called by an ICBM configuration routine to determine a controller letter.

Description

A device name is made up of three fields: ddxyy, where dd is the device name, x is the controller letter and yy is the unit number. Controller letters are assigned by Autoconfigure, starting with "A", based on the order in which devices are found. Certain letters are excluded from use. This routine provides a consistent way of assigning controller letters across the system.

The configuration routine should check the low byte of the busarray$l_ctrlltr field prior to calling this routine. If this value is non-zero, this value should be used as the controller letter. If this value is zero, then IOGEN$ASSIGN_CONTROLLER should be called to assign a controller letter. Note that the busarray$l_ctrlltr field is reserved to this routine and that *only* the low byte should be used by its callers.

IOGEN$AUTOCONFIGURE

Configures I/O devices found on the system.

Prototype

int iogen$autoconfigure (int flags, void * prefix, void * select, void * exclude, int * bus_list, log_callback( ), exclude_callback( ))

Parameters

Name	Access	Description
flags	Input	Controls optional autoconfigure behaviors follows: IOGEN$M_AC_LAN—Select all NI/FDDI devices. IOGEN$M_LOG—Log each device to SYS$OUTPU as it is configured. IOGEN$M_LOG_ALL—Log additional Autoconfigure progress/error messages. IOGEN$M_SCA–Select all SCA ports and related devices.
prefix	Input	Pointer to a descriptor containing the prefix list (obtained from IOGEN$GET_PREFIX).
select	Input	Pointer to a descriptor containing the select list (from the /SELECT qualifier)
exclude	Input	Pointer to a descriptor containing the exclude list (from the /EXCLUDE qualifier)
bus_list	Input	Pointer to an array of int containing a list of bus types to configure. This list is a zero-terminated list of longwords. Each longword represents the adapter type of a bus to be included in the configuration scan. Adapter bus types are specified using the AT$_ symbols defined in the dcdef.h header file, in SYS$STARTLET_C.TLB.

Name	Access	Description
log_callback	Input	Address of a routine to process log messages. If this routine is not specified, messages are logged to SYS$OUTPUT. Otherwise, this routine is called and passed a pointer to a string descriptor for the message to be logged.
exclude_callback	Input	Address of a routine to read the permanent exclusion list if the default routine will fail.

Return Values

SS$_NORMAL	Successfully autoconfigured the system.
SS$_ACCVIO	Ran out of workspace.
SS$_NOPRIV	Process does not have CMKRNL privilege any status returned by SYS$EXPREG system service.

Context

Called in user mode. Generally called to process the SYSMAN IO AUTOCONFIGURE command.

Description

This routine automatically configures the system.

IOGEN$GET_PREFIX

Read the prefix file SYS$MANAGER:IOGEN$PREFIX.DAT.

Prototype

int iogen$get_prefix (void * prefix)

Parameters

Name	Access	Description
prefix	Input	Address of a descriptor into which the prefix is returned.

Return Values

SS$_NORMAL	Successfully retrieved the prefix list.
SS$_IVBUFLEN	The descriptor is too short to hold the prefix list.

Context

User mode. Generally called by the SYSMAN IO SET PREFIX and IO SHOW PREFIX commands.

Description

This routine retrieves the prefix list from the file SYS$MANAGER: IOGEN$PREFIX.DAT. The prefix list is set by the SYSMAN IO SET PREFIX command.

By default, there is no prefix list (this routine returns a null string, which equates to a prefix list of "SYS$". When adding new prefixes, be sure to include the string "SYS$" in the list, otherwise, no devices will be configured except those configured by the ICBM identified by the new prefix.

IOGEN$LOG

Displays Autoconfiguration logging messages.

Prototype

int iogen$log (int handle, int status, void * device, void * driver, int noisy)

Parameters

Name	Access	Description
handle	Input	32-bit Autoconfiguration context. This value was passed into the ICBM configuration routine.
status	Input	The 32-bit status to be logged (usually a completion status from SYS$LOAD_DRIVER).
device	Input	Pointer to a descriptor containing the device name.
driver	Input	Pointer to a descriptor containing the driver name.
noisy	Input	Flag identifying a "noisy" message. Suppressed unless the LOG_ALL flag is set in the call to the IOGEN$AUTOCONFIGURE command.

Return Values

SS$_NORMAL	Successful completion.

Context

Executive mode. Called by an ICBM configuration routine to log success or failure configuring a device.

Description

This routine is called by an ICBM configuration routine to log the status of configuring the device. This routine will format and print a message, depending on whether the /LOG qualifier was specified on the SYSMAN IO AUTOCONFIGURE command.

SYS$LOAD_DRIVER

Load a device driver and configure I/O database for a device

Prototype

int sys$load_driver (int function, void * device, void * driver, int * itmlst, int * iosb, NULL, NULL, NULL)

Parameters

Name	Access	Description
function	Input	One of the function codes listed in Table 12–2.
device	Input	Pointer to a descriptor containing the device name.
driver	Input	Pointer to a descriptor containing the driver name. The default file specification SYS$LOADABLE_IMAGES:.EXE is applied to the driver name before loading it.
itmlst	Input	Pointer to an itemlist that qualifies the caller's request. This is a standard OpenVMS item list, and it is terminated by a longword of zero. See Table 12–3 for a list of available function codes.
iosb	Input	Pointer to an I/O status block which receives final request status.
efn	Input	Event flag.

Return Values

SS$_NORMAL	Request completed successfully.
SS$_BADPARAM	One or more parameters are incorrect.
SS$_IVBUFLEN	A buffer associated with an item list descriptor is not a valid length.
SS$_ACCVIO	A parameter was not available for the required access.
SS$_INSFMEN	Failure allocating non-paged pool.
SS$_NOCMKRNL	Process does not have CMKRNL privilege.

SS$_NOCMEXE	Process does not have CMEXEC privilege.
SS$_NOSYSLCK	Process does not have SYSLCK privilege.
SS$_NOSUCHDEV	Explicit controller/unit requested for non-existent device.
SS$_UNSUPORTED	The requested function is unsupported.
SS$_DRVEXISTS	The specified driver already exists.
SS$_DEVEXISTS	The specified device already exists.
SS$_DRV_NOUNLOAD	Driver is not unloadable.
SS$_ILLIOFUNC	Invalid function code.
SS$_UNSUPPORTED	Function not supported.

Context

Executive mode. Generally called by an ICBM configuration routine.

Description

This service is called to load a device driver and configure the I/O database for the new device. SYS$LOAD_DRIVER is called by an ICBM to load a device driver, and to create the I/O database structures for each new device unit.

SYS$LOAD_DRIVER requires information that the ICBM must locate in various structures in the I/O database created by system bus probing, notably the ADP and the Bus Array. Section 12.3.7 describes how to find this information for the buses supported on Digital platforms.

Table 12–2 Function Codes Available for `Function` Parameter

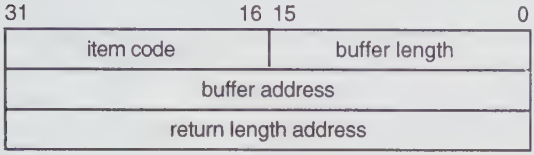
IOGEN$_LOAD	Analog to the SYSMAN IO LOAD command. Loads only the specified driver, ignoring the device and itmlst parameters. It applies the default SYS$LOADABLE_IMAGES:.EXE file specification.
IOGEN$_RELOAD	Analog to the SYSMAN IO RELOAD command. Reloadable drivers are not supported in OpenVMS Alpha, and this function returns the SS$_UNSUPPORTED error.

(continued on next page)

Table 12–2 (Cont.) Function Codes Available for `Function` Parameter

IOGEN$_ CONNECT	Analog to the SYSMAN IO CONNECT command. Creates all parts of the I/O database needed to allow the specified unit to operate, and connects the unit to the appropriate interrupt dispatching mechanism. The IOGEN$_LOAD, IOGEN$_INIT_CTRL, IOGEN$_INIT_UNIT functions are performed implicitly by the IOGEN$_CONNECT function. The device parameter is required. In many cases, an itmlst must be specified (see below). If the driver parameter is not specified, the name is defaulted by using the first two characters of the device name to locate the driver (for example, for device xxcu, the driver name is SYS$xxDRIVER).
IOGEN$_INIT_ CTRL	Causes the driver controller initialization routine to be called for the specified device. IPL is raised to 8 before calling the routine.
IOGEN$_INIT_ UNIT	Causes the driver unit initialization routine to be called for the specified device. IPL is raised to 8 before calling the routine.
IOGEN$_ DELIVER	Causes the unit delivery routine for the specified device to be called. The unit number passed to the delivery routine is specified in the itmlst parameter. IPL is raised to 8 before calling the delivery routine.

A single descriptor for an itemlist looks like this:



ZK-8682A-GE

Table 12–3 Function Codes Available for `Itemlist` Parameter

IOGEN$_ ADAPTER	Indicates to which nexus the specified device should be connected. A non-negative, 32-bit integer, and must match the `adp$w_tr` field of some ADP in the system. If the DDB for the specified device does not yet exist, either this item or the IOGEN$_NOADAPTER must be specified.
IOGEN$_ NOADAPTER	Indicates that the specified device is not associated with any particular bardware bus. This is usually the case for drivers that indicate the NULL adapter in their prologue tables.
IOGEN$_CSR	Specifies the CSR address for the device being configured. This value is stored in the `icb$q_csr` field. If the CRB for the device does not yet exist, and a non-null adapter is specified, this item is required.
IOGEN$_ VECTOR	Specifies the base interrupt vector for the device being configured. For directly vectored devices, this value is the byte offset into the SCB of the interrupt slot. For indirectly vectored devices, this value is the byte offset into the vector table pointed to by the associated ADP. If the CRB for the device does not yet exist, and a non-null adapter is specified, this item is required.

(continued on next page)

Table 12–3 (Cont.) Function Codes Available for `Itemlist` Parameter

IOGEN$_ NUMVEC	Specifies the number of consecutive interrupt vectors required by the specified device. The default value is 1. If a value greater than 1 is specified, the CRB is allocated such that it includes an `intd` field for each vector. These additional `intd` fields are connected to the SCB slots or the ADP vector table slots, as appropriate, following the base slot specified with IOGEN$_VECTOR.
IOGEN$_ MAXUNITS	Specifies the maximum number of units that can be associated with the specified controller. If this item is not specified, the value is taken from the driver's prologue table. This value determines the size of the `ucblst` field in the IDB. Note that the `ucblst` is not filled in; it is allocated and zeroed.
IOGEN$_SYSID	Specifies which remote system is the controller for the specified device. This value is a 64-bit integer that is the station address of the controller. This value is used to locate the appropriate system block. If this item is not specified, the specified device is assumed locally connected.
IOGEN$_UNIT	Specifies the unit number to be passed to the driver's unit delivery routine. Any unit number specified in the device parameter is ignored; instead, unit numbers are specified with this item code. This item must be specified if the function IOGEN$_DELIVER is invoked.
IOGEN$_ NUMUNITS	Specifies the number of units to be created on a single call to SYS$LOAD_DRIVER. The unit number specified in the device parameter is used for the first unit. Additional units are created with monotonically increasing unit numbers. This value is minimized with the value derived for IOGEN$_MAXUNITS. If this item is not specified, one unit is created. SYS$LOAD_DRIVER returns the number of units created in the buffer pointed to by this item descriptor.

(continued on next page)

Table 12–3 (Cont.) Function Codes Available for `Itemlist` Parameter

IOGEN$_ DELIVER_DATA	Specifies a 64-bit buffer that is passed as scratch space to the device driver's unit delivery routine. The actual data passed to the driver is allocated from non-paged pool and initialized from the buffer pointed to by this item descriptor. When the driver's unit delivery routine completes, the non-paged pool scratch area is copied back to the caller's buffer. If this item is omitted, the scratch area passed to the driver is initialized to zero.
IOGEN$_DDB	Specifies a buffer which receives the address of the DDB for the specified device. If no DDB exists, zero is returned.
IOGEN$_CRB	Specifies a buffer which receives the address of the CRB for the specified device. If no CRB exists, zero is returned.
IOGEN$_IDB	Specifies a buffer which receives the address of the IDB for the specified device. If no IDB exists, zero is returned.
IOGEN$_UCB	Specifies a buffer which receives the address of the UCB for the specified device. If no UCB exists, zero is returned.
IOGEN$_SB	Specifies a buffer which receives the address of the system block for the specified device. If no system block exists, zero is returned.

12.3.7 Finding Info in the Bus Array

When OpenVMS boots, system-specific routines probe all the I/O buses to locate hardware devices. A list of ADPs is created which describes this hardware. (See section 16.2.4.4 for one example.)

This section describes where the specific information needed by an ICBM is located for each bus. This information includes the following:

device ID	Used by an ICBM to identify devices.
adapter	Required by the IOGEN$_ADAPTER item.
CSR	Required by the IOGEN$_CSR item.
interrupt vector	Required by the IOGEN$_VECTOR item.
node	Required by the IOGEN$_NODE item.

For each bus, all of these except the device ID and the interrupt vector appear in the same place:

adapter	`adp$l_tr`
CSR	`busarray$q_csr`
node	`busarray$l_node_number`

The device ID always appears in BUSARRAY$Q_HW_ID. However, the format of the device ID is different for each bus type. The interrupt vectors also appear in different places for each bus type.

12.3.7.1 TURBOchannel

The device ID is the first 8 bytes of the TURBOchannel device option ROM. This field generally contains the option name. In the example ICBM, this field contains the ASCII string "AV300-AA".

The interrupt vector is located in `busarray$l_autoconfig`.

12.3.7.2 PCI

The device ID is a 32-bit value formed by combining the two 16-bit Vendor ID and Device ID values. These values are located in the first two fields (at offsets 0 and 2) of PCI configuration space for the device. The Vendor ID is in the high 16-bits; the device ID is in the low 16-bits.

The interrupt vector is located in `busarray$l_bus_specific_l`.

12.3.7.3 ISA

The device ID is the first 8-bytes of the value specified as the handle parameter in the system console's isacfg command. (See the hardware reference guide for your platform for details about the isacfg console command.) The system bus probing routine copies this value into BUSARRAY$Q_HS_ID.

The interrupt vector is located in busarray$l_bus_specific_l. The value located in this field must be multiplied by 4 before being passed to the IOGEN$_VECTOR item. The value in BUSARRAY$L_BUS_SPECIFIC_L is the same as the IRQ parameter specified in the ISACFG console command for the device.

Note that ISA devices can be loaded using the ISA_CONFIG.DAT mechanism, described in Chapter 15.

13

Debugging a Device Driver

The OpenVMS Delta and XDelta Debuggers (DELTA/XDELTA) and the OpenVMS Alpha System-Code Debugger (system-code debugger) are tools you can use to debug device drivers. This chapter briefly describes DELTA/XDELTA, and it explains how to use the system-code debugger.

13.1 Using the Delta/XDelta Debugger

The OpenVMS Delta/XDelta Debugger (DELTA/XDELTA) is a primitive debugger. It is used to debug code that cannot be debugged with the symbolic debugger, that is, any code that executes at interrupt priority levels (IPLs) above IPL0 or any code that executes in supervisor, executive, or kernel mode. Examples include user-written device drivers and the OpenVMS operating system.

Almost all the commands available on DELTA are also available on XDELTA. Furthermore, both DELTA and XDELTA use the same expressions. However, they are different in two ways: you use them to debug different kinds of code, and you invoke and exit from them in different ways.

You can use DELTA to debug programs that execute at IPL0 in any processor mode (user, supervisor, executive, and kernel). You can also debug user-mode programs with DELTA, but the OpenVMS Debugger is more suitable. To run DELTA in a processor mode other than user mode, your process must have the privilege that allows DELTA to change to that mode—change-mode-to-executive (CMEXEC) or change-mode-to-kernel (CMKRNL) privilege. You cannot use DELTA to debug code that executes at an elevated IPL.

You can use XDELTA to debug programs that execute in any processor mode and at any IPL. To use XDELTA, you must have system privileges, and you must include XDELTA when you boot the system.

You can use DELTA/XDELTA commands to perform the following debugging tasks:

- Open, display, and change the value of a particular location
- Set, clear, and display breakpoints
- Set display modes in byte, word, longword, or ASCII
- Display instructions
- Execute the program in a single step with the option to step over a subroutine
- Set base registers
- List the names and locations of all loaded modules of the executive
- Map an address to an executive module

For more information about using DELTA/XDELTA, see the *OpenVMS Delta /XDelta Debugger Manual*.

13.2 Using the OpenVMS Alpha System-Code Debugger

The OpenVMS Alpha System-Code Debugger (system-code debugger) can be used to debug nonpageable system code and device drivers running at any interupt priority level (IPL). You can use the system-code debugger to perform the following tasks:

- Control the system software's execution—stop at points of interest, resume execution, intercept fatal exceptions, and so on
- Trace the execution path of the system software
- Monitor exception conditions
- Examine and modify the values of variables
- In some cases, test the effect of modifications without having to edit the source code, recompile, and relink

The system-code debugger is a symbolic debugger. You can specify variable names, routine names, and so on, precisely as they appear in your source code. The system-code debugger can also display the source code where the software is executing, and allow you to step by source line.

The system-code debugger recognizes the syntax, data typing, operators, expressions, scoping rules, and other constructs of a given language. If your code or driver is written in more than one language, you can change the debugging context from one language to another during a debugging session.

To use the system-code debugger, you must do the following:

- Build a system image or device driver to be debugged.
- Set up the target kernel on a standalone system.

  The **target kernel** is the part of the system-code debugger that resides on the system that is being debugged. It is integrated with XDELTA and is part of the SYSTEM_DEBUG execlet.
- Set up the host system, which is integrated with the OpenVMS Debugger.

The following sections cover these tasks in more detail, describe the available user-interface options, summarize applicable OpenVMS Debugger commands, and provide a sample system-code debugger session.

13.2.1 User-interface Options

The system-code debugger has the following user-interface options:

- A DECwindows Motif interface for workstations

  When using this interface, you interact with the system-code debugger by using a mouse and pointer to choose items from menus, click on buttons, select names in windows, and so on.

  Note that you can also use OpenVMS Debugger commands with the DECwindows Motif interface.
- A character cell interface for terminals and workstations

  When using this interface, you interact with the system-code debugger by entering commands at a prompt. The sections in this chapter describe how to use the system-code debugger with the character cell interface.

For more information about using the OpenVMS DECwindows Motif interface and OpenVMS Debugger commands with the system-code debugger, see the *OpenVMS Debugger Manual*.

13.2.2 Building a System Image to Be Debugged

1. Compile the sources you want to debug, and be sure to use the /DEBUG and /NOOPT qualifers.

   Note

   Debugging optimized code is much more difficult and is not recommended unless you know the Alpha architecture well. The instructions are reordered so much that single-stepping by source line will look like you are randomly jumping all over the code. Also note

that you cannot access all variables. The system-code debugger reports that they are optimized away.

---

2. Link your image using the the /DSF (debug symbol file) qualifier. Do not use the /DEBUG qualifier, which is for debugging user programs. The /DSF qualifier takes an optional filename argument similar to the /EXE qualifier. For more information, see the *OpenVMS Linker Utility Manual*. If you specify a name in the /EXE qualifier you will need one for the /DSF qualifier. For example you would use the following command:

   ```
   $ LINK/EXE=EXE$:MY_EXECLET/DSF=EXE$:MY_EXECLET OPTIONS_FILE/OPT
   ```

   The .DSF and .EXE file names should be the same. Only the extensions will be different, that is .DSF and .EXE.

   The contents of the .EXE file should be exactly the same as if you had linked without the /DSF qualifier. The .DSF file will be a small image containing the image header and all the debug symbol tables for that image. It is not an executable file, so you should not try to run it or load it.

3. Put the .EXE file on your target system.

4. Put the .DSF file on your host system, because when you use the system-code debugger to debug code in your image, it will try to look for a .DSF file first and then look for a .EXE file. The .DSF file is better because it has symbols in it. Section 13.2.4 describes how to tell the system-code debugger where to find your .DSF and .EXE files.

13.2.3 Setting Up the Target System for Connections

The target kernel is controlled by flags and devices specified when the system is booted, XDELTA commands, a configuration file, and several sysgen parameters. The following sections contain more information about these items.

Boot Command

The form of the boot command varies depending on the type of OpenVMS Alpha system you are using. However, all boot commands have the concept of boot flag and boot devices as well as a way to save the default boot flags and devices. This section uses syntax from a DEC 3000 Model 400 Alpha Workstation in examples.

To use the system-code debugger, you must specify an Ethernet device with the boot command on the target system. This device will be used by the target system to communicate with the host debugger. It is currently a restriction that this device must not be used for anything else (either for booting or

network software such as DECnet, TCP/IP products, and LAT products). Thus, you must also specify a different device to boot from. For example, the following command will boot a DEC 3000 Model 400 from the dkb100 disk, and the system-code debugger will use the esa0 ethernet device.

```
>>> boot dkb100,esa0
```

To find out the Ethernet devices available on your system, enter the following command:

```
>>> Show Device
```

In addition to devices, you can also specify flags on the boot command line. Boot flags are specified as a hex number; each bit of the number represents a true or false value for a flag. The following flag values are relevant to the system-code debugger:

- **8000**

  This new boot flag enables operation of the target kernel. If this boot flag is not set, not only will it be impossible to use the system-code debugger to debug the system, but the additional XDELTA commands related to the target kernel will generate an XDELTA error message. If this flag is set, SYSTEM_DEBUG is loaded, and the system-code debugger is enabled.

- **0004**

  This boot flag's function has not changed. It controls whether the system calls INI$BRK at the beginning and end of EXEC_INIT. Notice that if the system-code debugger is the default debugger, the first breakpoint is not as early as it is for XDELTA. It is delayed until immediately after the PFN database is set up.

- **0002**

  This boot flag, which has always controlled whether XDELTA is loaded, behaves slightly differently when the system-code debugger boot flag is also set.

  If the system-code debugger boot flag is clear, this flag works as it always has. If the system-code debugger boot flag is set, this flag determines whether XDELTA or the system-code debugger is the default debugger. If the XDELTA flag is set, XDELTA will be the default debugger. In this state, the initial system breakpoints and any calls to INI$BRK trigger XDELTA, and you must enter an XDELTA command to start using the system-code debugger. If this flag is clear, the initial breakpoints and calls to INI$BRK go to the system-code debugger. You cannot use XDELTA if the XDELTA flag is clear.

Boot Command Example The following command boots a DEC 3000 Model 400 from the dka0 disk, enables the system-code debugger, defaults to using XDELTA, and takes the initial system boot breakpoints:

```
>>> boot dka0,esa0 -fl 0,8006
```

You can set these devices and flags to be the default values so that you will not have to specify them each time you boot the system. On a DEC 3000 Model 400, use the following commands:

```
>>> set BOOTDEF_DEV dka0,esa0
>>> set BOOT_OSFLAGS 0,8006
```

System-Code Debugger Configuration File

The system-code debugger target system reads a configuration file in SYS$SYSTEM named DBGTK$CONFIG.SYS. The first line of this file contains a default password, which must be specified by the host debug system to connect to the target. Other lines in this file are reserved by Digital. Note that you must create this file because Digital does not supply it. If this file does not exist, you cannot run the system-code debugger.

XDELTA Commands

When the system is booted with the 8000 boot flag, the following two additional XDELTA commands are enabled:

- n,\xxxx;R ContRol System-Code Debugger connection

  This command can be used to do the following:

  - Change the password which the system-code debugger must present
  - Disconnect the current session from the system-code debugger
  - Give control to the remote debugger by simulating a call to INI$BRK
  - Any combination of these

  Optional string argument xxxx specifies the password that the system-code debugger must present for its connection to be accepted. If this argument is left out, the required password is unchanged. The initial password is taken from the first line of the SYS$SYSTEM:DBGTK$CONFIG.SYS file.

  The optional integer argument *n* controls the behavior of the ;R command as follows:

Value of N	Action
+1	Gives control to the system-code debugger by simulating a call to INI$BRK
+2	Returns to XDELTA after changing the password. 2;R without a password is a no-op
0	Performs the default action
-1	Changes the password, breaks any existing connection to the System Debugger and then simulates a call to INI$BRK (which will wait for a new connection to be established and then give control to the system-code debugger)
-2	Returns to XDELTA after changing the password and breaking an existing connection

Currently, the default action is the same action as +1.

If the system-code debugger is already connected, the ;R command transfers control to the system-code debugger, and optionally changes the password that must be presented the next time a system-code debugger tries to make a connection. This new password does not last across a boot of the target system.

- n;K Change inibrK behavior

  If optional argument *n* is 1, future calls to INI$BRK will result in a breakpoint being taken by the system-code debugger. If the argument is 0, or no argument is specified, future calls to INI$BRK will result in a breakpoint being taken by XDELTA.

Sysgen Parameters

- **DBGTK_SCRATCH**

  This new parameter specifies how many pages of memory are allocated for the system-code debugger. This memory is allocated only if system-code debugging is enabled with the 8000 boot flag (described earlier in this section). Usually, the default value of 1 is adequate; however, if the system-code debugger displays an error message, increase this value.

- **SCSNODE**

  Identifies the target kernel node name for the system-code debugger. See Section 13.2.3.1 for more information.

13.2.3.1 Making Connections Between the Target Kernel and the System-Code Debugger

It is always the system-code debugger that initiates a connection to the target kernel. When the system-code debugger initiates this connection, the target kernel accepts or rejects the connection based on whether the remote debugger presents it with a node name and password that matches the password in the target system (either the default password from the SYS$SYSTEM:DBGTK$CONFIG.SYS file, or a different password specified via XDELTA). The system-code debugger gets the node name from the SCSNODE Sysgen parameter.

The target kernel can accept a connection from the system-code debugger anytime the system is running below IPL 22, or if XDELTA is in control (at IPL 31) However, the target kernel actually waits at IPL 31 for a connection from the system-code debugger in two cases: When it has no existing connection to a system-code debugger and (1) It receives a breakpoint caused by a call to INI$BRK (including either of the initial breakpoints), or (2) when you enter a 1;R or -1;R command from XDELTA.

13.2.3.2 Interactions between XDELTA and the Target Kernel/System-Code Debugger

XDELTA and the target kernel are integrated into the same system. Normally, you choose to use one or the other. However, XDELTA and the target kernel can be used together. This section explains how they interoperate.

The 0002 boot flag controls which debugger (XDELTA or the target kernel) gets control first. If it is not set, the target kernel gets control first, and it is not possible to use XDELTA without rebooting. If it is set, XDELTA gets control first, but you can use XDELTA commands to switch to the system-code debugger and to switch INI$BRK behavior such that the system-code debugger gets control when INI$BRK is called.

Breakpoints always *stick* to the debugger that set them. For example, if you set a breakpoint at location "A" with XDELTA, and then you enter the command 1;K (switch INI$BRK to the system-code debugger) and ;R (start using the system-code debugger). Then, from the system-code debugger, you set a breakpoint at location "B". If the system executes the breakpoint at A, XDELTA will report a breakpoint, and the remote debugger will see nothing (though you could switch the system-code debugger by issuing the XDELTA ;R command). If the system executes the breakpoint at B, the system-code debugger will get control and report a breakpoint (you cannot switch to XDELTA).

Notice that if you examine location A with the system-code debugger, or location B with XDELTA, you will see a BPT instruction, not the instruction that was originally there. This is because neither debugger has any information about the breakpoints set by the other debugger.

One useful way to use both debuggers together is when you have a system that exhibits a failure only after hours or days of heavy use. In this case, you can boot the system with the system-code debugger enabled (8000), but with XDELTA the default (0002) and with initial breakpoints enabled (0004). When you reach the initial breakpoint, set an XDELTA breakpoint at a location that will only be reached when the error occurs. Then proceed. When the error breakpoint is reached, possibly days later, then you can set up a remote system to debug it and enter the ;R command to XDELTA to switch control to the system-code debugger.

Here is another technique for use when you do not know where to put an error breakpoint as previously mentioned. Boot the system with only the 8000 flag. When you see that the error has happened, halt the system and initiate an IPL 14 interrupt, as you would to start XDELTA. The target kernel will get control and wait for a connection for the system-code debugger.

13.2.4 Setting Up the Host System

To set up the host system, you need access to all system images and drivers that are loaded (or can be loaded) on the target system. You should have access to a source listings kit or a copy of the following directories:

SYS$LOADABLE_IMAGES:
SYS$LIBRARY:
SYS$MESSAGE:

You need all the .EXE files in those directories. The .DSF files are available with the OpenVMS Alpha source listings kit.

Optionally, you need access to the source files for the images to be debugged. The system-code debugger will look for the source files in the directory where they were compiled. If your build machine and host machine are different, you must use the SET SOURCE command to point the system-code debugger to location of the source code files. For an example of the SET SOURCE command, see Section 13.4.2.

To make the connection, you must set up the logical DBGHK$IMAGE_PATH, which must be set up as a search list to the area where the system images are kept. For example, if the copies are in the following directories,

```
DEVICE:[SYS$LDR]
DEVICE:[SYSLIB]
DEVICE:[SYS$MSG]
```

you would define DBGHK$IMAGE_PATH as follows:

```
$ define dbghk$image_path DEVICE:[SYS$LDR],DEVICE:[SYSLIB],DEVICE:[SYS$MSG]
```

This works well for debugging using all the images normally loaded on a given system. However, you might be using the debugger to test new code in an execlet or a new driver and might want to debug that new code. Because that image is most likely in your default directory, you must define the logical as follows:

```
$ define dbghk$image_path [],DEVICE:[SYS$LDR],DEVICE:[SYSLIB],DEVICE:[SYS$MSG]
```

If the system-code debugger cannot find one of the images through this search path, a warning message is displayed. The system-code debugger will continue initialization as long as it finds at least one image. If the system-code debugger cannot find the SYS$BASE_IMAGE file, which is the OpenVMS Alpha operating system's main image file, an error message is displayed and the debugger exits.

Check the directory for the image files and compare it to what is loaded on the target system.

13.2.5 Starting the System-Code Debugger

To start the system-code debugger on the host side, enter the following command:

```
$ DEBUG/KEEP
```

The system-code debugger displays the DBG> prompt. With the DBGHK$IMAGE_PATH logical defined, you can invoke the CONNECT command and optional qualifiers /PASSWORD and /IMAGE_PATH.

To use the CONNECT command and optional qualifiers (/PASSWORD and /IMAGE_PATH) to connect to the node with name <node-name> enter the following command:

```
DBG> CONNECT %NODE_NAME node-name /PASSWORD="password"
```

If a password has been set up on the target system, you must use the /PASSWORD qualifier. If a password is not specified, a zero length string is passed to the target system as the password.

The /IMAGE_PATH qualifier is also optional. If you do not use this qualifier, the system-code debugger uses the DBGHK$IMAGE_PATH logical as the default. The /IMAGE_PATH qualifier is a quick way to change the logical. However, when you use it, you cannot specify a search list. You can use only a logical or a device and directory, although the new logical could be a search list.

Usually, the system-code debugger gets the file name from the object file. This is put there by the compiler when the source is compiled with the /DEBUG qualifier. The SET SOURCE command can take a list of paths as a parameter. It treats them as a search list.

13.2.6 Summary of OpenVMS Debugger Commands

The following OpenVMS Debugger commands can also be used with the system-code debugger:

DISPLAY
EXPAND
EXTRACT
MOVE
SAVE
SCROLL
SEARCH
SELECT
TYPE
SET MARGINS
SET SEARCH
SET WINDOW
CANCEL DISPLAY
CANCEL WINDOW
SHOW DISPLAY
SHOW MARGINS
SHOW SEARCH
SHOW SELECT
SHOW WINDOW

The following commands are useful for writing OpenVMS Debugger command programs and for adding new commands at run time:

DECLARE
DO
EXITLOOP
FOR
IF
REPEAT
WHILE

The following commands are useful for miscellaneous operations:

ATTACH
CTRL_C
CTRL_Y

EDIT
SPAWN
SET ABORT_KEY
SET ATSIGN
SET EDITOR
SET KEY
SET LOG
SET MODE
SET OUTPUT
SET PROMPT
SET RADIX
SET TERMINAL
CANCEL MODE
CANCEL RADIX
SHOW ABORT_KEY
SHOW ATSIGN
SHOW EDITOR
SHOW KEY
SHOW LOG
SHOW MODE
SHOW OUTPUT
SHOW RADIX
SHOW TERMINAL

The following commands manipulate symbols and source code for the debugged code. For example, you can use these commands to define new symbols, change the current scope (to a different image, module or routine), tell the debugger where the source code resides, and set the current language. The SHOW IMAGE command behaves like the XDELTA ;L command. The DEFINE command behaves in a similar way to the XDELTA ;X command.

DEFINE
DELETE
EVALUATE
SYMBOLIZE
SET DEFINE
SET IMAGE
SET LANGUAGE
SET SCOPE
SET MAX_SOURCE_FILES
SET SOURCE
SET TYPE
CANCEL IMAGE
CANCEL MODULE

```
CANCEL SCOPE
CANCEL SOURCE
CANCEL TYPE
SHOW DEFINE
SHOW IMAGE
SHOW LANGUAGE
SHOW SCOPE
SHOW MAX_SOURCE_FILES
SHOW SOURCE
SHOW SYMBOL
SHOW TYPE
```

The following commands make the user program (or the system-code debugger) execute code. The GO command is the same as the XDELTA ;P and ;G commands (GO takes an optional PC value). The STEP command is the same as the XDELTA S and O commands for single stepping. These commands are implemented in both the main and kernel sections of the debugger.

```
GO
RERUN
RUN
STEP
SET STEP
SHOW STEP
```

The following commands set, cancel, show and temporarily deactivate and activate breakpoints. These commands are the same as the XDELTA ;B command. However, unlike, XDELTA there is no limit on the number of breakpoints.

```
SET BREAK
CANCEL BREAK
SHOW BREAK
CANCEL ALL
ACTIVATE BREAK
DEACTIVATE BREAK
```

The following commands access the user programs' (or in this case the system-code debugger) memory and registers. The DEPOSIT and EXAMINE commands implement the following set of XDELTA commands: /, !, [, ", '.

```
DEPOSIT
EXAMINE
SHOW STACK
SHOW CALLS
```

13.2.7 System-Code Debugger Network Information

The system-code debugger and the target kernel on the target system use a private Ethernet protocol to communicate. For the two systems to see each other, they have to be on the same Ethernet segment.

The network portion of the target system finds the first Ethernet device and communicates through it. The network portion of the host system also finds the first Ethernet device and communicates through it. However, if for some reason, the system-code debugger picks the wrong device, you can override this by defining the logical DBGHK$ADAPTOR to the template device name for the appropriate adaptor.

13.3 Troubleshooting Checklist

If you have trouble starting a connection, perform the following tasks to correct the problem:

- Check SCSNODE on the target system.

  It must match the name you are using in the host CONNECT command.

- Make sure that both the Ethernet and boot device are on the boot command.

- Make sure that the host system is using the correct Ethernet device, and that the host and target systems are connected to the same Ethernet segment.

- Check the version of the operating system and make sure that both the host and target systems are running the same version of the OpenVMS Alpha operation gsystem.

13.4 Troubleshooting Network Failures

There are three possible network errors:

- NETRERTRY

  Displayed if the system-code debugger connection is lost.

- SENDRETRY

  Indicates a message send failure.

- NETFAIL

  Caused by the two previous messages.

The netfail error message has a status code that can be one of the following values:

Value	Status
2, 4, 6	Internal network error, submit an SPR with the code.
8,10,14,16,18,20,26,28,34,38	Network protocol error, submit an SPR with the code.
22,24	Too many errors on the network device most likely due to congestion. Reduce the network traffic or switch to another network backbone.
30	Target system scratch memory not available. Check DBGTK_SCRATCH. If increasing this value does not help, submit an SPR.
32	Ran out of target system scratch memory. Increase value of DBGTK_SCRATCH.
All others	There should not be any other network error codes printed. If one occurs that does not match the above, submit an SPR.

13.4.1 Access to Symbols in OpenVMS Executive Images

Accessing OpenVMS executive images' symbols is not always straightforward with the system-code debugger. Only a subset of the symbols may be accessible at one time and in some cases, the symbol value the debugger currently has may be stale. To understand these problems and their solutions, you must understand how the debugger maintains its symbol tables and what symbols exist in the OpenVMS executive images. The following sections briefly summarize these topics.

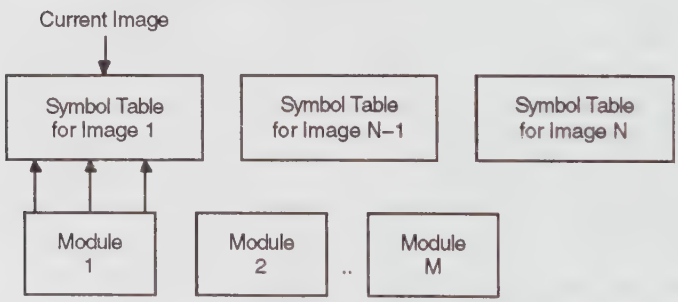
13.4.1.1 Overview of How the OpenVMS Debugger Maintains Symbols

The debugger can access symbols from any image in the OpenVMS loaded system image list by either reading in the .DSF or .EXE file for that particular image. The .EXE file only contains information about symbols that are part of the symbol vector for that image. The current image symbols for any set module are defined. (You can tell if you have the .DSF or .EXE by doing a SHOW MODULE. If there are no modules you have the .EXE.) This includes any symbols in the SYS$BASE_IMAGE.EXE symbol vector for which the code or data resides in the current image. However, a user cannot access a symbol that is part of the SYS$BASE_IMAGE.EXE symbol vector that resides in another image.

In general, at any one point in time, the debugger can only access the symbols from one image. (A later section describes how to get around this limitation). It does this to reduce the time it takes to search for a symbol in a table. To load the symbols for a particular image, use the SET IMAGE command. When you set an image, the debugger loads all the symbols from the new image and makes that image the current image. The symbols from the previous image are in memory but the debugger will not look through it to translate symbols. To remove symbols from memory for an image, use the CANCEL IMAGE command (which does not work on the main image, SYS$BASE_IMAGE).

There is a set of modules for each image the debugger accesses. The symbol tables in the image that are part of these modules are not loaded with the SET IMAGE command. Instead they can be loaded with the SET MODULE module-name or SET MODULE/ALL commands. As they are loaded, a new symbol table is created in memory under the symbol table for the image. The following figure shows what this looks like:

Figure 13–1 Maintaining Symbols



So, when the debugger needs to translate a value into a symbol or a symbol into a value, it first looks at the current image to find the information. If it does not find it there, then it looks into the appropriate module. It determines which module is appropriate by looking at the module range symbols which are part of the image symbol table.

To see what symbols are currently loaded, use the DEBUG SHOW SYMBOL command. This command has a few options to get more than just the symbol name and value. See the *OpenVMS Debugger Manual* for more details.

13.4.1.2 Overview of OpenVMS Executive Image Symbols

Depending on whether the debugger has access to the .DSF or .EXE file, different kinds of symbols could be loaded. Most users will have the .EXE file for the OpenVMS executive images and a .DSF file for their private images—that is, the images they are debugging.

The OpenVMS executive consists of two base images, SYS$BASE_IMAGE.EXE and SYS$PUBLIC_VECTORS.EXE, and a number of separately loadable executive images.

The two base images contain symbol vectors. For SYS$BASE_IMAGE.EXE the symbol vector is used to define symbols accessible by all the separately loadable images. This allows these images to communicate with each other through cross-image routine calls and memory references. For SYS$PUBLIC_VECTORS.EXE, the symbol vector is used to define the OpenVMS system services. Because these symbol vectors are in the .EXE and the .DSF files, the debugger can load these symbols no matter which one the user has.

All images in the OpenVMS executive also contain global and local symbols. However, none of these symbols ever get into the .EXE file for the image. These symbols are put in the specific modules section of the .DSF file if that module was compiled /DEBUG and the image was linked /DSF.

13.4.1.3 Possible Problems You May Encounter

- **Access to All Executive Image Symbols**

  When the current image is not SYS$BASE_IMAGE, but one of the separately loaded images, the debugger does not have access to any of the symbols in the SYS$BASE_IMAGE symbol vector. This means the user cannot access (set break points, etc.) any of the cross-image routines or data cells. The only symbols the user has access to are the ones defined by the current image.

  If the debugger only has access to the .EXE file, this means no symbols at all for images with no symbol vectors. For .DSF files, the current image symbols for any set module are defined. (You can tell if you have the .DSF or .EXE by using the SHOW MODULE command—if there are no modules you have the .EXE). This includes any symbols in the SYS$BASE_IMAGE.EXE symbol vector for which the code or data resides in the current image. However, the user cannot access a symbol that is part of the SYS$BASE_IMAGE.EXE symbol vector that resides in another image. For example, if you are in one image and you want to set a break point in a cross-image routine from another image, you do not have access to the symbol. Of course, if you know which image it is defined in, you can just do a SET IMAGE, SET MODULE/ALL and then a SET BREAK.

There is a debugger workaround for this problem. The debugger and the system-code debugger let you use the SET MODULE command on an image by prefixing the image name with SHARE$ (SHARESYSBASE_IMAGE for example). This treats that image as a module which is part of the current image. In the previous figure, think of it as another module in the module list for an image. Note, however, that only the symbols for the symbol vector are loaded. None of the symbols for the modules of the SHARE$xxx image are loaded. Therefore, this command is only useful for base images.

So in other words, by doing SET MODULE SHARESYSBASE_IMAGE, the debugger gives you access to all cross-image symbols for the VMS Executive.

- **Stale Data From the Symbol Vector**

  When an OpenVMS Executive Based Image is loaded, the values in the symbol vectors are only correct for information that resides in that based image. For all symbols that are defined in the separately loaded images, it contains a pointer to a placeholder location. For routine symbols this is a routine that just returns an image not loaded failure code. A symbol vector entry is fixed to contain the real symbol address when the image in which the data resides is loaded.

  Therefore, if the user does a SET IMAGE to a base image before all the symbol entries are corrected, it will get the placeholder value for those symbols. Then once the image containing the real data is loaded, the debugger will still have the placeholder value. This means the user is looking at stale data. One solution to this is to make sure to do a CANCEL IMAGE and SET IMAGE on the based image in order to get the most up to date symbol vector loaded into memory.

  The CANCEL IMAGE/SET IMAGE combination does not currently work for SYS$BASE_IMAGE because it is the main image and DEBUG does not allow you to CANCEL the main image. Therefore, if you connect to the target system early in the boot process, you will have stale data as part of the SYS$BASE_IMAGE symbol table. However, the SET MODULE SHARE$xxx command always re-loads the information from the symbol vector. So to get around this problem you could SET IMAGE to an image other than SYS$BASE_IMAGE and then do use CANCEL MODULE SHARESYSBASE_IMAGE and SET MODULE SHARESYSBASE_IMAGE to do the same thing. The only other solution is to always connect to the target system once all images are loaded which define the real data for values in the symbol vectors. You could also enter the following commands, and you would get the latest values from in the symbol vector.

```
SET IMAGE EXEC_INIT
SET MODULE/ALL
SET MODULE SHARE$SYS$BASE_IMAGE
```

- **Problems with SYS$BASE_IMAGE.DSF**

  For those that have access to the SYS$BASE_IMAGE.DSF file, there may be another complication with accessing symbols from the symbol vector. The problem is that the module SYSTEM_ROUTINES contains the placeholder values for each symbol in the symbol vector. So, if SYSTEM_ROUTINES is the currently set module (which is the case if you are sitting at the INI$BRK break point) then the debugger will have the placeholder value of the symbol as well as the value in the symbol vector. You can see what values are loaded with the SHOW SYMBOL/ADDRESS command. The symbol vector version should be marked with (global), the local one is not.

  To set a break point at the correct code address for a routine when in this state, use the SHOW SYMBOL/ADDRESS command on the routine symbol name. If the global and local values for the code address are the same, then the image with the routine has not been loaded yet. If not, set a break point at the code address for the global symbol.

13.4.2 Sample System-Code Debugging Session

This section provides a sample session that shows the use of some DEBUG commands as they apply to the system-code debugger. The examples in this session show how to work with C code that has been linked into the SYSTEM_DEBUG execlet. It is called as an initialization routine for SYSTEM_DEBUG.

To reproduce this sample session, you need access to the SYSTEM_DEBUG.DSF matching the SYSTEM_DEBUG.EXE file on your target system and to the source file C_TEST_ROUTINES.C, which is available in SYS$EXAMPLES. The target system is booted with the command bootflags 0, 8004 DKA0, ESA0, so it stops at an initial breakpoint.

The example begins by invoking the system-code debugger's character cell interface.

Example 13–1 Invoking the System-Code Debugger

```
$ define dbg$decw$display " " ! Don't use Motif version
$ debug/keep

        OpenVMS Alpha Alpha DEBUG Version T2.0-001

DBG>
```

Use the CONNECT command to connect to the target system. In this example, a password is not set up, and the example uses the logical DBGHK$IMAGE_PATH for the image path; those qualifiers are not being used. You may need to use them.

When you have connected to the target system, the DEBUG prompt is displayed. Enter the SHOW IMAGE command to see what has been loaded. Because you are reaching a breakpoint early in the boot process, there are very few images. See Example 13–2. Notice that SYS$BASE_IMAGE has an asterisk next to it. This is the currently set image and all symbols currently loaded in the debugger come from that image.

Example 13–2 Connecting to the Target System

```
DBG> connect %node_name TSTSYS
%DEBUG-I-EXPMEMPOOL, expanding debugger memory pool
DBG> sho image

 image name                      set    base address    end address

 ERRORLOG                        no     00000000        0000E000
    NPRO0                               8005C000        8005EE00
    NPRW1                               80830200        80830800
 EXEC_INIT                       no     8234C000        82366000
*SYS$BASE_IMAGE                  yes    00000000        00028000
    NPRO0                               80002000        8000CA00
    NPRW1                               80804000        8081EA00
 SYS$CNBTDRIVER                  no     00000000        0000C000
    NPRO0                               8000E000        8000F400
    NPRW1                               8081EA00        8081EE00
 SYS$CPU_ROUTINES_0402           no     00000000        00016000
    NPRO0                               80060000        80068E00
    NPRW1                               80830800        80833200
 SYS$OPDRIVER                    no     00000000        00030000
    NPRO0                               80010000        80013C00
    NPRW1                               8081EE00        8081F800
 SYS$PUBLIC_VECTORS              no     00000000        00008000
    NPRO0                               80000000        80001600
    NPRW1                               80800000        80804000
 SYSTEM_DEBUG                    no     00000000        00034000
    NPRO0                               80014000        80034C00
    NPRW1                               8081F800        80827C00
 SYSTEM_PRIMITIVES               no     00000000        0002A000
    NPRO0                               80036000        80050200
    NPRW1                               80827C00        8082E400
 SYSTEM_SYNCHRONIZATION          no     00000000        00016000
    NPRO0                               80052000        8005BA00
    NPRW1                               8082E400        80830200

 total images: 10                bytes allocated: 517064
```

Example 13–3 shows the console display during the connect sequence. Note that for security reasons, the name of the host system, the user's name, and process ID are displayed.

Example 13–3 Target System Connection Display

```
DBGTK: Initialization succeeded.  Remote system debugging is now possible.
DBGTK: Waiting at breakpoint for connection from remote host.
DBGTK: Connection attempt from host HSTSYS   user GUEST       process 45800572
DBGTK: Connection attempt succeeded
```

Example 13–4 Setting a Breakpoint

```
DBG> set image system_debug
DBG> show module
module name                    symbols    size

C_TEST_ROUTINES                no         2152
FATAL_EXC                      no         3116
SERVER_NET                     no         2632
TARGET_KERNEL                  no        18296

total C modules: 5.            bytes allocated: 549256.
DBG> set module c_test_routines
DBG> show module
module name                    symbols    size

C_TEST_ROUTINES                yes        2152
FATAL_EXC                      no         3116
SERVER_NET                     no         2632
TARGET_KERNEL                  no        18296

total C modules: 5.            bytes allocated: 553848.
DBG> set language c
DBG> show symbol test_c_code*
routine C_TEST_ROUTINES\test_c_code3
routine C_TEST_ROUTINES\test_c_code2
routine C_TEST_ROUTINES\test_c_code
DBG> set break test_c_code
DBG> sho break
breakpoint at routine C_TEST_ROUTINES\test_c_code
```

To set a breakpoint at the first routine in the C_TEST_ROUTINES module of the SYSTEM_DEBUG.EXE execlet, do the following:

1. Load the symbols for the SYSTEM_DEBUG image with the DEBUG SET IMAGE command.
2. Use the SET MODULE command to get the symbols for the module.
3. Set the language to be C and set a breakpoint at the routine test_c_code.

The language must be set because C is case sensitive and test_c_code needs to be specified in lower case. The language is normally set to the language of the main image, in this example SYS$BASE_IMAGE.EXE. Currently that is not C.

Now that the breakpoint is set, you can proceed and activate the breakpoint. When that occurs, the debugger tries to open the source code for that location in the same place as where the module was compiled. Because that is not the same place as on your system, you need to tell the debugger where to find the source code. This is done with the DEBUG SET SOURCE command, which takes a search list as a parameter so you can make it point to many places.

Example 13–5 Finding the Source Code

```
DBG> go
break at routine C_TEST_ROUTINES\test_c_code
%DEBUG-W-UNAOPNSRC, unable to open source file DSKD$:[DELTA.SRC]C_TEST_ROUTINES.
C;*
-RMS-F-DEV, error in device name or inappropriate device type for operation
    80: Source line not available
DBG> set source sys$examples:
```

Now that the debugger has access to the source, you can put the debugger into screen mode to see exactly where you are and the code surrounding it.

Example 13–6 Using the Set Mode Screen Command

```
DBG> Set Mode Screen; Set Step Nosource

- SRC: module C_TEST_ROUTINES -scroll-source-----------------------------------
    63:     if (xdt$fregsav[9] > 0)
    64:         *pVar =  (*pVar + xdt$fregsav[17])%xdt$fregsav[9];
    65:     else
    66:         *pVar = (*pVar + xdt$fregsav[17]);
    67:     xdt$fregsav[7] = test_c_code3(10);
    68:     xdt$fregsav[3] = test;
    69:     return xdt$fregsav[23];
    70: }
    71: void test_c_code(void)
    72: {
    73:     int x,y;
    74:     int64 x64,y64;
    75:
->  76:     x = xdt$fregsav[0];
    77:     y = xdt$fregsav[1];
    78:     x64 = xdt$fregsav[2];
    79:     y64 = xdt$fregsav[3];
    80:     xdt$fregsav[14] = test_c_code2(x64+y64,x+y,x64+x,&y64);
    81:     return;
    82: }
- OUT -output------------------------------------------------------------------



- PROMPT -error-program-prompt-------------------------------------------------
```

(continued on next page)

Example 13–6 (Cont.) Using the Set Mode Screen Command

```
DBG>
```

Now, you want to set another breakpoint inside the test_c_code3 routine. You use the SCROLL/UP DEBUG command (8 on the keypad) to move to that routine and see that line 56 would be a good place to set the breakpoint. It is at a recursive call. Then you proceed to that breakpoint with the GO command.

Example 13–7 Using the SCROLL/UP DEBUG Command

```
- SRC: module C_TEST_ROUTINES -scroll-source-----------------------------------
    44: Source line not available
    45: Source line not available
    46: Source line not available
    47: #pragma noinline(test_c_code,test_c_code2,test_c_code3)
    48: extern volatile int64 xdt$fregsav[34]; /* Lie and say these are integer
    49:                                        use iregsav because the debugger will r
    50:                                        be using those!*/
    51:
    52: int test_c_code3(int subrtnCount)
    53: {
    54:     subrtnCount = subrtnCount - 1;
    55:     if (subrtnCount != 0)
    56:         subrtnCount = test_c_code3(subrtnCount);
    57:     return subrtnCount;
    58: }
    59: int test_c_code2(int64 in64,int in32, int64 test, int64* pVar)
    60: {
    61:     xdt$fregsav[5] = in64;
    62:     xdt$fregsav[6] = in32;
    63:     if (xdt$fregsav[9] > 0)
    64:         *pVar =  (*pVar + xdt$fregsav[17])%xdt$fregsav[9];
    65:     else
- OUT -output-----------------------------------------------------------------




- PROMPT -error-program-prompt-----------------------------------------------

DBG> Scroll/Up
DBG> set break %Line 56
DBG> go
DBG>
```

When you reach that breakpoint, the source code display is updated to show where you currently are, which is indicated by an arrow. A message also appears in the OUT display indicating you reach the breakpoint at that line.

Example 13–8 Break Point Display

```
- SRC: module C_TEST_ROUTINES ------------------------------------------------
    46: Source line not available
    47: #pragma noinline(test_c_code,test_c_code2,test_c_code3)
    48: extern volatile int64 xdt$fregsav[34]; /* Lie and say these are integer
    49:                                        use iregsav because the debugger will r
    50:                                        be using those!*/
    51:
    52: int test_c_code3(int subrtnCount)
    53: {
    54:     subrtnCount = subrtnCount - 1;
    55:     if (subrtnCount != 0)
->  56:         subrtnCount = test_c_code3(subrtnCount);
    57:     return subrtnCount;
    58: }
    59: int test_c_code2(int64 in64,int in32, int64 test, int64* pVar)
    60: {
    61:     xdt$fregsav[5] = in64;
    62:     xdt$fregsav[6] = in32;
    63:     if (xdt$fregsav[9] > 0)
    64:         *pVar =  (*pVar + xdt$fregsav[17])%xdt$fregsav[9];
    65:     else
    66:         *pVar = (*pVar + xdt$fregsav[17]);
    67:     xdt$fregsav[7] = test_c_code3(10);
- OUT -output----------------------------------------------------------------
break at C_TEST_ROUTINES\test_c_code3\%LINE 56









- PROMPT -error-program-prompt-----------------------------------------------


DBG> Scroll/Up
DBG> set break %Line 56
DBG> go
DBG>
```

Now you try the DEBUG STEP command. The default behavior for STEP is STEP/OVER unlike XDELTA and DELTA which is STEP/INTO. So normally you would expect to step to line 57 in the code. However, because you have a breakpoint inside test_c_code3 that is called at line 56, you will reach that event first.

Example 13–9 Using the Debug Step Command

```
- SRC: module C_TEST_ROUTINES -scroll-source-----------------------------------
    46: Source line not available
    47: #pragma noinline(test_c_code,test_c_code2,test_c_code3)
    48: extern volatile int64 xdt$fregsav[34]; /* Lie and say these are integer
    49:                                        use iregsav because the debugger will r
    50:                                        be using those!*/
    51:
    52: int test_c_code3(int subrtnCount)
    53: {
    54:     subrtnCount = subrtnCount - 1;
    55:     if (subrtnCount != 0)
->  56:         subrtnCount = test_c_code3(subrtnCount);
    57:     return subrtnCount;
    58: }
    59: int test_c_code2(int64 in64,int in32, int64 test, int64* pVar)
    60: {
    61:     xdt$fregsav[5] = in64;
    62:     xdt$fregsav[6] = in32;
    63:     if (xdt$fregsav[9] > 0)
    64:         *pVar =  (*pVar + xdt$fregsav[17])%xdt$fregsav[9];
    65:     else
    66:         *pVar = (*pVar + xdt$fregsav[17]);
    67:     xdt$fregsav[7] = test_c_code3(10);
- OUT -output-----------------------------------------------------------------
break at C_TEST_ROUTINES\test_c_code3\%LINE 56
break at C_TEST_ROUTINES\test_c_code3\%LINE 56




- PROMPT -error-program-prompt-----------------------------------------------

DBG> Scroll/Up
DBG> set break %Line 56
DBG> go
DBG> step
DBG>
```

Now, you try a couple of other commands, EXAMINE and SHOW CALLS. The EXAMINE command allows you to look at all the C variables. Note that the C_TEST_ROUTINES module is compiled with the /NOOPTIMIZE switch which allows access to all variables. The SHOW CALLS command shows you the call sequence from the beginning of the stack. In this case, you started out in the

image EXEC_INIT. (The debugger prefixes all images other than the main image with SHARE$ so it shows up as SHARE$EXEC_INIT.)

Example 13–10 Using the Examine and Show Calls Commands

```
- SRC: module C_TEST_ROUTINES -scroll-source-----------------------------------
    46: Source line not available
    47: #pragma noinline(test_c_code,test_c_code2,test_c_code3)
    48: extern volatile int64 xdt$fregsav[34]; /* Lie and say these are integer
    49:                                        use iregsav because the debugger will r
    50:                                        be using those!*/
    51:
    52: int test_c_code3(int subrtnCount)
    53: {
    54:     subrtnCount = subrtnCount - 1;
    55:     if (subrtnCount != 0)
->  56:         subrtnCount = test_c_code3(subrtnCount);
    57:     return subrtnCount;
    58: }
    59: int test_c_code2(int64 in64,int in32, int64 test, int64* pVar)
    60: {
    61:     xdt$fregsav[5] = in64;
    62:     xdt$fregsav[6] = in32;
    63:     if (xdt$fregsav[9] > 0)
    64:         *pVar =  (*pVar + xdt$fregsav[17])%xdt$fregsav[9];
    65:     else
    66:         *pVar = (*pVar + xdt$fregsav[17]);
    67:     xdt$fregsav[7] = test_c_code3(10);
- OUT -output-------------------------------------------------------------------
break at C_TEST_ROUTINES\test_c_code3\%LINE 56
break at C_TEST_ROUTINES\test_c_code3\%LINE 56
C_TEST_ROUTINES\test_c_code3\subrtnCount:       8
 module name     routine name                    line       rel PC    abs PC
*C_TEST_ROUTINES test_c_code3                     56       0000002C  8002A7CC
*C_TEST_ROUTINES test_c_code3                     56       0000003C  8002A7DC
*C_TEST_ROUTINES test_c_code2                     67       000000AC  8002A8A4
*C_TEST_ROUTINES test_c_code                      80       00000084  8002A960
                                                           00000000  8234A244
                                                           00000000  8234A0C0
 SHARE$EXEC_INIT                                           00000000  82379BC4


- PROMPT -error-program-prompt-------------------------------------------------
DBG> Scroll/Up
DBG> set break %Line 56
DBG> go
DBG> step
DBG> examine subrtnCount
DBG> show calls
DBG>
```

If you want to proceed because you are done debugging this code, first cancel all the breakpoints and then enter the GO command. Notice however, that you do not keep running but get a message that you have stepped to line 57. This happens because the STEP command used earlier never completed. It was interrupted by the breakpoint on line 56.

Note that the debugger remembers all step events and only removes them once they have completed.

Example 13–11 Canceling the Breakpoints

```
- SRC: module C_TEST_ROUTINES -scroll-source-----------------------------------
    47: #pragma noinline(test_c_code,test_c_code2,test_c_code3)
    48: extern volatile int64 xdt$fregsav[34]; /* Lie and say these are integer
    49:                                        use iregsav because the debugger will r
    50:                                        be using those!*/
    51:
    52: int test_c_code3(int subrtnCount)
    53: {
    54:     subrtnCount = subrtnCount - 1;
    55:     if (subrtnCount != 0)
    56:         subrtnCount = test_c_code3(subrtnCount);
->  57:     return subrtnCount;
    58: }
    59: int test_c_code2(int64 in64,int in32, int64 test, int64* pVar)
    60: {
    61:     xdt$fregsav[5] = in64;
    62:     xdt$fregsav[6] = in32;
    63:     if (xdt$fregsav[9] > 0)
    64:         *pVar =  (*pVar + xdt$fregsav[17])%xdt$fregsav[9];
    65:     else
    66:         *pVar = (*pVar + xdt$fregsav[17]);
    67:     xdt$fregsav[7] = test_c_code3(10);
    68:     xdt$fregsav[3] = test;
- OUT -output-------------------------------------------------------------------
break at C_TEST_ROUTINES\test_c_code3\%LINE 56
break at C_TEST_ROUTINES\test_c_code3\%LINE 56
C_TEST_ROUTINES\test_c_code3\subrtnCount:       8
 module name     routine name                    line       rel PC    abs PC
*C_TEST_ROUTINES test_c_code3                     56       0000002C  8002A7CC
*C_TEST_ROUTINES test_c_code3                     56       0000003C  8002A7DC
*C_TEST_ROUTINES test_c_code2                     67       000000AC  8002A8A4
*C_TEST_ROUTINES test_c_code                      80       00000084  8002A960
                                                           00000000  8234A244
                                                           00000000  8234A0C0
 SHARE$EXEC_INIT                                           00000000  82379BC4
stepped to C_TEST_ROUTINES\test_c_code3\%LINE 57

- PROMPT -error-program-prompt-------------------------------------------------
DBG> go
DBG> step
DBG> examine subrtnCount
DBG> show calls
DBG> cancel break/all
DBG> go
DBG>
```

Next, enter a STEP command at a return statement. Returns are branches on OpenVMS Alpha; however the debugger treats them as special cases. For branches the default is STEP/OVER; however for return instructions the default is STEP/INTO. You move up a level and are now at an event point at line 56.

The reason you are at line 56 and not line 57 is that you have returned from the subroutine; however you have not stored the result in subrtnCount yet.

Example 13–12 Using the Step Command

```
- SRC: module C_TEST_ROUTINES -scroll-source-----------------------------------
    46: Source line not available
    47: #pragma noinline(test_c_code,test_c_code2,test_c_code3)
    48: extern volatile int64 xdt$fregsav[34]; /* Lie and say these are integer
    49:                                        use iregsav because the debugger will r
    50:                                        be using those!*/
    51:
    52: int test_c_code3(int subrtnCount)
    53: {
    54:     subrtnCount = subrtnCount - 1;
    55:     if (subrtnCount != 0)
->  56:         subrtnCount = test_c_code3(subrtnCount);
    57:     return subrtnCount;
    58: }
    59: int test_c_code2(int64 in64,int in32, int64 test, int64* pVar)
    60: {
    61:     xdt$fregsav[5] = in64;
    62:     xdt$fregsav[6] = in32;
    63:     if (xdt$fregsav[9] > 0)
    64:         *pVar =  (*pVar + xdt$fregsav[17])%xdt$fregsav[9];
    65:     else
    66:         *pVar = (*pVar + xdt$fregsav[17]);
    67:     xdt$fregsav[7] = test_c_code3(10);
- OUT -output-------------------------------------------------------------------
break at C_TEST_ROUTINES\test_c_code3\%LINE 56
break at C_TEST_ROUTINES\test_c_code3\%LINE 56
C_TEST_ROUTINES\test_c_code3\subrtnCount:       8
 module name     routine name                     line       rel PC    abs PC
*C_TEST_ROUTINES test_c_code3                      56      0000002C  8002A7CC
*C_TEST_ROUTINES test_c_code3                      56      0000003C  8002A7DC
*C_TEST_ROUTINES test_c_code2                      67      000000AC  8002A8A4
*C_TEST_ROUTINES test_c_code                       80      00000084  8002A960
                                                           00000000  8234A244
                                                           00000000  8234A0C0
 SHARE$EXEC_INIT                                           00000000  82379BC4
stepped to C_TEST_ROUTINES\test_c_code3\%LINE 57
stepped to C_TEST_ROUTINES\test_c_code3\%LINE 56

- PROMPT -error-program-prompt-------------------------------------------------
DBG> step
DBG> examine subrtnCount
DBG> show calls
DBG> cancel break/all
DBG> go
DBG> step
DBG>
```

The STEP/RETURN command, a different type of step command, single steps assembly code until it finds a return instruction. This command is useful if you want to see the return value for the routine, which is done here by examining the R0 register.

For more information about using other STEP command qualifiers, see the *OpenVMS Debugger Manual*. For other useful STEP qualifiers, see the DEBUG documentation for more details.

Example 13–13 Using the Step/Return Command

```
- SRC: module C_TEST_ROUTINES -scroll-source-----------------------------------
    47: #pragma noinline(test_c_code,test_c_code2,test_c_code3)
    48: extern volatile int64 xdt$fregsav[34]; /* Lie and say these are integer
    49:                                        use iregsav because the debugger will r
    50:                                        be using those!*/
    51:
    52: int test_c_code3(int subrtnCount)
    53: {
    54:     subrtnCount = subrtnCount - 1;
    55:     if (subrtnCount != 0)
    56:         subrtnCount = test_c_code3(subrtnCount);
->  57:     return subrtnCount;
    58: }
    59: int test_c_code2(int64 in64,int in32, int64 test, int64* pVar)
    60: {
    61:     xdt$fregsav[5] = in64;
    62:     xdt$fregsav[6] = in32;
    63:     if (xdt$fregsav[9] > 0)
    64:         *pVar =  (*pVar + xdt$fregsav[17])%xdt$fregsav[9];
    65:     else
    66:         *pVar = (*pVar + xdt$fregsav[17]);
    67:     xdt$fregsav[7] = test_c_code3(10);
    68:     xdt$fregsav[3] = test;
- OUT -output-------------------------------------------------------------------
break at C_TEST_ROUTINES\test_c_code3\%LINE 56
C_TEST_ROUTINES\test_c_code3\subrtnCount:       8
 module name     routine name                    line       rel PC    abs PC
*C_TEST_ROUTINES test_c_code3                     56      0000002C  8002A7CC
*C_TEST_ROUTINES test_c_code3                     56      0000003C  8002A7DC
*C_TEST_ROUTINES test_c_code2                     67      000000AC  8002A8A4
*C_TEST_ROUTINES test_c_code                      80      00000084  8002A960
                                                          00000000  8234A244
                                                          00000000  8234A0C0
 SHARE$EXEC_INIT                                          00000000  82379BC4
stepped to C_TEST_ROUTINES\test_c_code3\%LINE 57
stepped to C_TEST_ROUTINES\test_c_code3\%LINE 56
stepped on return from C_TEST_ROUTINES\test_c_code3\%LINE 56+16 to C_TEST_ROUTI
C_TEST_ROUTINES\test_c_code3\%R0:       0
- PROMPT -error-program-prompt-------------------------------------------------
DBG> show calls
DBG> cancel break/all
DBG> go
DBG> step
DBG> step/return
DBG> examine r0
DBG>
```

(continued on next page)

Example 13–13 (Cont.) Using the Step/Return Command

After you finish the system-code debugging session, enter the GO command to leave this module. You will encounter another INI$BRK breakpoint at the end of EXEC_INIT. An error message indicating there are no source lines for address 80002010 is displayed, because debug information on this image or module is not available. The debugger leaves the source code for C_TEST_ROUTINES on the screen; however, it is not valid.

Also notice that there is no message in the OUT display for this event. That is because INI$BRKs are special breakpoints that are handled as SS$_DEBUG signals. They are a method for the system code to break into the debugger and there is no real breakpoint in the code.

Example 13–14 Source Lines Error Message

```
- SRC: module C_TEST_ROUTINES -scroll-source-----------------------------------
%DEBUG-W-SCRNOSRCLIN, no source line for address 80002010 for display in SRC
    45: Source line not available
    46: Source line not available
    47: #pragma noinline(test_c_code,test_c_code2,test_c_code3)
    48: extern volatile int64 xdt$fregsav[34]; /* Lie and say these are integer
    49:                                        use iregsav because the debugger will r
    50:                                        be using those!*/
    51:
    52: int test_c_code3(int subrtnCount)
    53: {
    54:     subrtnCount = subrtnCount - 1;
    55:     if (subrtnCount != 0)
    56:         subrtnCount = test_c_code3(subrtnCount);
->  57:     return subrtnCount;
    58: }
    59: int test_c_code2(int64 in64,int in32, int64 test, int64* pVar)
    60: {
    61:     xdt$fregsav[5] = in64;
    62:     xdt$fregsav[6] = in32;
    63:     if (xdt$fregsav[9] > 0)
    64:         *pVar =  (*pVar + xdt$fregsav[17])%xdt$fregsav[9];
    65:     else
--------------------------------------------------------------------------------
break at C_TEST_ROUTINES\test_c_code3\%LINE 56
C_TEST_ROUTINES\test_c_code3\subrtnCount:       8
```

(continued on next page)

Example 13–14 (Cont.) Source Lines Error Message

```
 module name     routine name                     line       rel PC    abs PC
*C_TEST_ROUTINES test_c_code3                      56      0000002C  8002A7CC
*C_TEST_ROUTINES test_c_code3                      56      0000003C  8002A7DC
*C_TEST_ROUTINES test_c_code2                      67      000000AC  8002A8A4
*C_TEST_ROUTINES test_c_code                       80      00000084  8002A960
                                                           00000000  8234A244
                                                           00000000  8234A0C0
 SHARE$EXEC_INIT                                           00000000  82379BC4
stepped to C_TEST_ROUTINES\test_c_code3\%LINE 57
stepped to C_TEST_ROUTINES\test_c_code3\%LINE 56
stepped on return from C_TEST_ROUTINES\test_c_code3\%LINE 56+16 to C_TEST_ROUTI
C_TEST_ROUTINES\test_c_code3\%R0:       0
- PROMPT -error-program-prompt-------------------------------------------------
DBG> cancel break/all
DBG> go
DBG> step
DBG> step/return
DBG> examine r0
DBG> go
DBG>
```

If you enter GO, the target system boots completely, because there are no more breakpoints in the boot path. The debugger will wait for another event to occur.

If you enter the SHOW IMAGE command, more images are displayed.

Example 13–15 Using the Show Image Command

```
- SRC: module C_TEST_ROUTINES -scroll-source-----------------------------------
%DEBUG-W-SCRNOSRCLIN, no source line for address 80002010 for display in SRC
    45: Source line not available
    46: Source line not available
    47: #pragma noinline(test_c_code,test_c_code2,test_c_code3)
    48: extern volatile int64 xdt$fregsav[34]; /* Lie and say these are integer
    49:                                        use iregsav because the debugger will r
    50:                                        be using those!*/
    51:
    52: int test_c_code3(int subrtnCount)
    53: {
    54:     subrtnCount = subrtnCount - 1;
    55:     if (subrtnCount != 0)
    56:         subrtnCount = test_c_code3(subrtnCount);
->  57:     return subrtnCount;
    58: }
    59: int test_c_code2(int64 in64,int in32, int64 test, int64* pVar)
    60: {
    61:     xdt$fregsav[5] = in64;
    62:     xdt$fregsav[6] = in32;
    63:     if (xdt$fregsav[9] > 0)
    64:         *pVar =  (*pVar + xdt$fregsav[17])%xdt$fregsav[9];
    65:     else
-------------------------------------------------------------------------------
    NPRW0                                  80852C00        80853000
    PRO1                                   824AA000        824ADE00
    PRW2                                   824AE000        824AE600
*SYSTEM_DEBUG                   yes        00000000        00034000
    NPRO0                                  80028000        80048C00
    NPRW1                                  80823200        8082B600
 SYSTEM_PRIMITIVES              no         00000000        0002A000
    NPRO0                                  8004A000        80064200
    NPRW1                                  8082B600        80831E00
 SYSTEM_SYNCHRONIZATION         no         00000000        00016000
    NPRO0                                  80066000        8006FA00
    NPRW1                                  80831E00        80833C00

 total images: 43               bytes allocated: 713656
- PROMPT -error-program-prompt-------------------------------------------------
DBG> go
DBG> step
DBG> step/return
DBG> examine r0
DBG> go
DBG> show image
DBG>
```

Part IV

Bus Support Information

Part IV describes bus-specific and processor-specific details that affect the composition and operation of OpenVMS Alpha device drivers. It includes the following chapters:

- Chapter 14 discusses Peripheral Component Interconnect (PCI) bus concepts and implementations on Alpha platforms.
- Chapter 15 describes ISA device configuration on OpenVMS Alpha systems that support ISA as an I/O bus.
- Chapter 16 describes Extended Industry Standard Architecture (EISA) bus concepts and implementations on OpenVMS Alpha platforms.

14

PCI Bus Support

This chapter discusses Peripheral Component Interconnect (PCI) bus concepts and implementations on Alpha platforms.

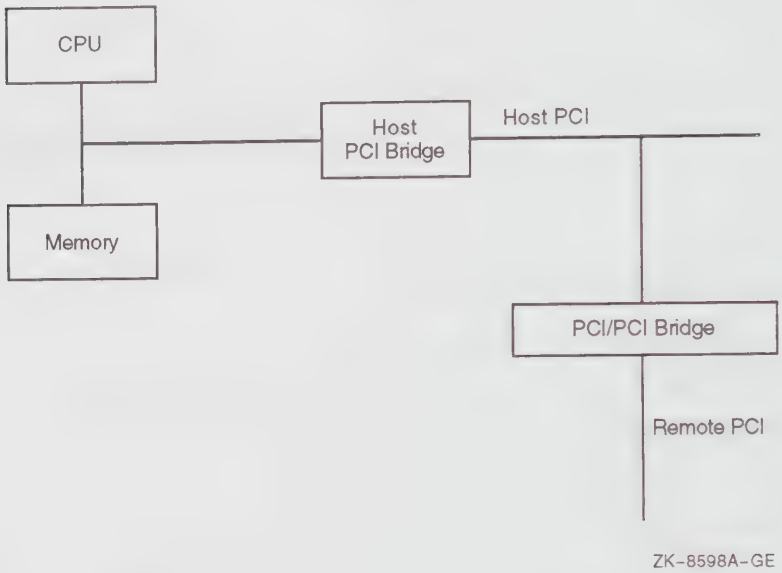
PCI bus characteristics include the following:

- 32-bit address space with optional 64-bit addressing capability
- 32-bit data path with optional 64-bit data path
- Separate address spaces (see Section 14.1.)
- Synchronous bus operation at frequencies up to 33 MHz

  33 MHz operation yields 132 MB/second peak performance: 33 * 10^6 cycles/second * 4 bytes/cycle = 132 MB/second.

A PCI bus is designed to accomodate multiple levels of buses. The PCI bus closest to the CPU is accessed through a host-to-PCI bridge, which is called the **host PCI**. **Remote PCI** buses are accessed through PCI-to-PCI bridge chips, which are connected to PCI buses closer to the processor. Figure 14–1 illustrates a generic PCI-based platform configuration.

Figure 14–1 PCI-Based Platform



14.1 PCI Address Spaces

The *PCI Specification* (as described by the PCI Special Interest Group) defines three separate 32-bit address spaces: configuration, I/O, and memory.

- PCI configuration space is intended for use primarily during booting and configuration, although it is required to be accessible at all times.
- PCI I/O space is generally used for registers and control functions that require byte and word length access. It is similar to EISA I/O space.
- PCI memory space is intended for devices with memory buffers that require memory address space, such as frame buffers. It is also intended for device registers. However, 64-bit addressing and 64-bit data transfers are also defined for PCI memory space.

Digital platforms use a combination of different physical address regions and CSR control bits to permit access to all three spaces. On the hardware level, address spaces can be accessed by using different PCI transaction types. For example, for a read to memory space, the hardware generates a Mem_Read cycle. For a read to I/O space, the hardware generates an IO_Read cycle. And for a read to configuration space, the hardware generates a Config_Read cycle.

The following subsections contain more details about these PCI address spaces. For more information about the location and size of PCI physical address spaces, see the appropriate system map in Appendix A.

14.1.1 PCI Configuration Space

Every PCI device has its own section of configuration space address space. Within this configuration space the device must implement a predefined header (called the **configuraton space header**) that is accessible at offset 0 in the device configuration space. The configuration space header contains the following information:

Vendor identification number
Device identification number
Device class and type
Base address registers

To determine which PCI devices are present in the system, PCI bus probing routines attempt to read the vendor and device identification numbers from the configuration space header of each potential PCI slot on the host-to-PCI bus.

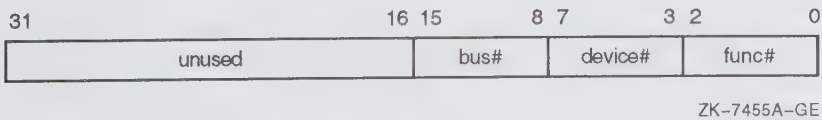
The *PCI specification* defines a mechanism for accessing the configuration space of all possible PCI devices, whether the devices are on the host PCI or on a remote PCI. This mechanism encodes the following information to form a unique configuration space address:

Bus number (0-255, where bus 0 is always the host PCI CPU)
Device number (0-31)
Function number (0-7)

The device number is analogous to a backplane slot number, although it is really decoded by hardware into a chip select signal for a single PCI device. Therefore, a PCI bus can be treated as a "slot-based" bus, which means that a device can be found from the bus number and the device slot number.

To match the PCI specification's definition of a configuration space address, OpenVMS Alpha defines a PCI node number as shown in Figure 14–2.

Figure 14–2 PCI Node Number



Although 5 bits are required for the device number, electrical loading considerations usually limit the number of PCI devices on a bus to less than 32 devices.

The *PCI specification* defines up to 6 base address registers in the configuration space header. The base address registers are used to locate the device in the proper PCI address space (memory or I/O). Bus mapping software reads a base address register to determine how much and what kind of address space a device requires and then assigns the base address of the device by writing a PCI physical address to the base address register. On Alpha systems, the console assigns PCI address spaces.

A device may implement up to 6 base address registers. This allows a device to use up to 6 separate address ranges for device registers or memory buffers. Generally a device will only require one or two base address registers.

The predefined configuration space header and the base address registers enable system independent software to locate all PCI devices in the system address space, and to assign address space to devices in a conflict-free configuration.

14.1.2 PCI I/O Space

Access to PCI I/O space is through a swizzle space address encoding with a 5-bit address shift. Only a small portion of the GB PCI I/O space is addressible by the CPU (due to the 5-bit address swizzle). Some platforms allow access to 128 MB of PCI I/O space, while others may allow access to only the lowest 64 KB of PCI I/O space. Lack of addressibility of the entire PCI I/O address space is not seen as a problem because PCI devices are encouraged to implement device registers in PCI memory space, and INTEL processors can only access 64 KB of PCI I/O space.

14.1.3 PCI Memory Space

PCI memory space is accessible in both dense space and swizzle space. There are separate platform physical address regions for swizzle space and dense space. The access characteristics of each space are different. Swizzle space (5 bit address shift) is intended for byte, word, long, and quad access granularity. The size of the transfer and which bytes will be transfered are encoded in bits 6:3 of the CPU address. Software must align the data in the correct byte lanes. This means that bytes will always appear in their natural byte lanes, based upon byte address. To maintain ordering of data transfers, software must issue memory barriers after each device access. Note that the platform independent access routines IOC$READ_IO or IOC$WRITE_IO contain the necessary instructions. Device control registers that are implemented in PCI memory space should be assigned to swizzle space using the console.

The minimum access granularity of dense space is a longword. In dense space, the Alpha CPU address maps directly to the PCI address—there is no address bit shifting as in swizzle space. Platforms are permitted to implement read prefetching and write merging in dense space. Device control registers should not be placed in dense space. Dense space is intended for frame buffers and other on-chip buffers with memory-like behavior.

14.2 PCI Device Interrupts

The *PCI Specification* does not define an interrupt mechanism for I/O device interrupts. Some systems implement interrupts using PC style interrupt controller chips, such as the 8259. Other systems implement custom interrupt handling logic.

In general, a distinction is made between motherboard PCI devices, which are built into the system and always present, and option slot devices. A motherboard device generally interrupts through a unique input on the system interrupt controller.

For PCI option slots, the *PCI Specification* defines 4 interrupt signals per slot, called INTA, INTB, INTC, and INTD. There are few rules about how systems are supposed to present option slot interrupts to the system interrupt logic. Some systems combine the INTx signals from each slot and present a single slot interrupt to the system interrupt logic. Other systems present each INTx signal as a unique interrupt to the system interrupt logic.

14.3 OpenVMS Alpha PCI Bus Support Data Structures

A PCI bus is represented by an Adapter Control Block (ADP) and an associated bus array. The bus array has an entry for each PCI device attached to the bus. The ADP address and PCI node number allow software to find the bus array entry associated with a PCI device.

For more information about the ADP data structure, see Chapter 17.

14.4 Direct Memory Access (DMA) on the PCI Bus

Direct Memory Access (DMA) refers to PCI devices reading or writing system memory. The PCI bus places no restrictions on DMA. From the point of view of a PCI device, system memory can be viewed as another device on the PCI with an assigned address space. A PCI device performs DMA operations by issuing reads or writes to the address space assigned to system memory.

However, Alpha platform implementations place some restrictions on how a PCI device must perform DMA. There are a number of reasons for these restrictions, mostly related to EISA and ISA compatibility.

Current Alpha platforms define at least two PCI DMA windows, which can be thought of as the PCI memory address space assigned to system memory.

You can check the platform address map for each platform to see how the DMA windows are set up on that machine. All platforms with a PCI bus will have at least one scatter/gather PCI DMA window. For maximum driver portability, you should code your driver to use the scatter/gather map for DMA. In OpenVMS Alpha, scatter/gather support (also known as map register support) is accomplished through the use of the generic counted resource management routines IOC$ALLOC_CRCTX, IOC$ALLOC_CNT_RES, and IOC$LOAD_MAP. For detailed information about these routines, see Chapter 19.

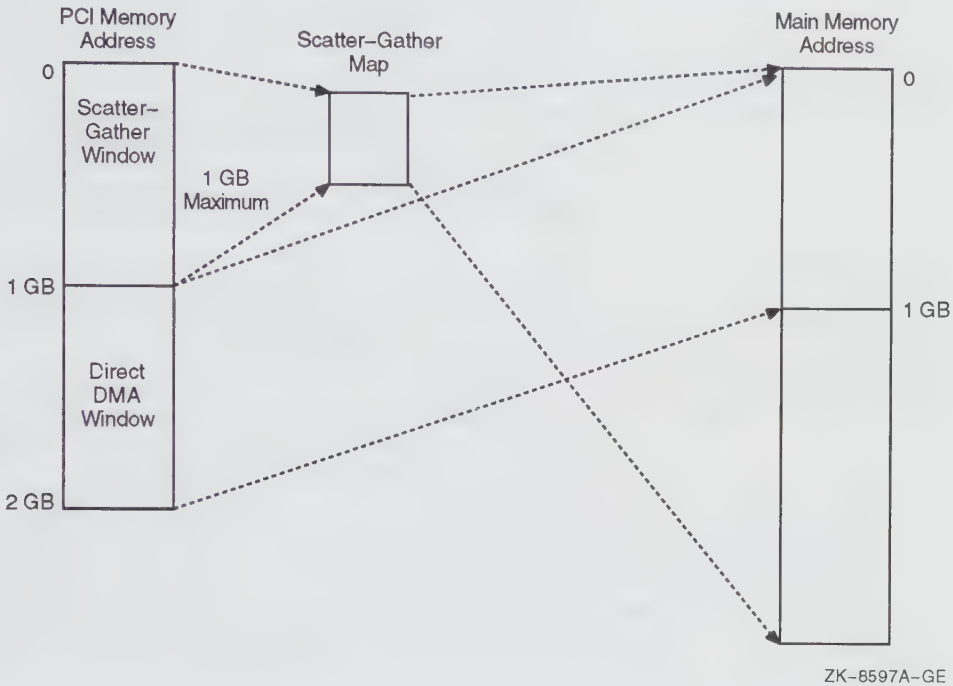
PCI bus implementations on Alpha platforms define two ways for PCI devices to access main memory: scatter/gather memory access and physical memory access. In **scatter/gather memory access**, the PCI address generated by a PCI device is translated to a main memory address by a scatter/gather table. In **physical memory access** the PCI address generated by a PCI device is translated to a main memory address by the addition of a constant. Scatter/gather memory access is called **scatter/gather DMA**. Physical memory access is called **physical (or direct) DMA**.

Each memory access technique has advantages and disadvantages. Scatter /gather DMA allows access to all system memory, but it is more complex to program. Physical DMA is easier to program, but it may limit the amount of main memory that can be addressed.

Alpha PCI platforms implement scatter/gather DMA and physical DMA through **DMA address windows** on the PCI bus. A PCI DMA address window is an address range on the PCI bus through which PCI devices (and EISA/ISA devices behind the PCI/EISA bridge or PCI/ISA bridge) access main memory. The Alpha platforms that support PCI buses (AlphaServer 2000, AlphaServer 2100, AlphaServer 1000, AlphaServer 400, AlphaServer 8200, AlphaServer 8400, AlphaStation 200, AlphaStation 250) provide a minimum of two DMA address windows on the PCI bus. A DMA address window can be a physical DMA window, where a PCI bus address is a linear function of a system memory address, or a scatter/gather DMA window, where a PCI bus address undergoes a page table translation before becoming a main memory address.

Figure 14–3 shows a typical platform, in which a maximum of 2 GB of the PCI memory address space maps to main memory. A PCI address in the range 1GB to 2 GB, the direct DMA window, is combined with a base register to produce a main memory address between 0 and 1GB.

Figure 14–3 Example of PCI Memory Address Space Maped to Main Memory



Alpha platforms that support PCI buses have both a physical DMA window and a scatter/gather DMA window. In OpenVMS Alpha Version 6.2, the scatter /gather DMA window is based at PCI address 0 and extends to a maximum address of 3FFFFFFF (the actual size of the scatter/gather DMA window is a function of the amount of physical memory in the system). In OpenVMS Alpha Version 6.2, the physical DMA window is typically based at a PCI address above the scatter/gather DMA window.

The IOC$NODE_DATA routine includes two function codes that allow drivers to find the base of the physical DMA window in a platform independent manner. The format of the IOC$NODE_DATA routine is as follows:

```
int ioc$node_data (CRB  *crb, int  function_code, void *user_buffer)
```

Inputs:

crb	Address of CRB. IOC$NODE_DATA uses the crb$l_node and the vec$ps_adp fields to find the data structures associated with the I/O bus to which this device is connected.
function_code	From iocdef.h in SYS$LIB_C.TLB. Specifies information to be returned by IOC$NODE_DATA.
user_buffer	Address of caller's buffer. On success, the requested information is returned in the caller's buffer.

Outputs:

SS$_NORMAL	Normal successful completion. Requested information is returned in the caller's buffer.
SS$_ILLIOFUN	Unrecognized function code.
SS$_BADPARAM	Bad parameter. Usually this means that crb$l_node contains an invalid slot number. Check that the driver has been loaded with the /node qualifier.

When called with function code IOC$K_DIRECT_DMA_BASE, the IOC$NODE_DATA routine returns the base address of the physical DMA window. The base address is returned as a 64-bit value in anticipation of future 64-bit I/O buses. For this reason, the caller of the IOC$NODE_DATA routine should make sure the user_buffer argument points to a quadword cell when using the IOC$K_DIRECT_DMA_BASE function code.

When called with function code IOC$K_DIRECT_DMA_SIZE, the IOC$NODE_DATA routine returns the size of the physical DMA window, expressed in megabytes. The size of the direct DMA window is returned as a 32-bit value.

Using the physical DMA window for device DMA is straightforward. Once you have found the base of the physical DMA window using the IOC$NODE_DATA routine, you must adjust the DMA address that you assign to the device by adding the physical DMA window base to the main memory DMA buffer address. For example, on a typical system the physical DMA window is based at the PCI address 40000000 and extends to 7FFFFFFF. PCI addresses in the range from 40000000 to 7FFFFFFF are passed to main memory addresses 0 to 3FFFFFFF. This means that if you have a DMA buffer at main memory address 0, the PCI device would access this buffer at PCI address 40000000. The correspondence of main memory address and PCI DMA addresses is derived by the following formula:

```
PCI DMA address = (main memory buffer address) + (base of physical DMA window)
```

Note that the size of the physical DMA window limits the amount of physical memory that can be addressed by a PCI, EISA, or ISA device. In the physical DMA window, it is not possible for an I/O device to address more than 1 GB of physical memory. For this reason, on large memory systems you may want to code your driver to perform DMA in the scatter/gather window. To perform scatter/gather DMA, use the counted resource management routines described in the Chapter 5.

14.5 Probing a PCI Bus to Find Devices

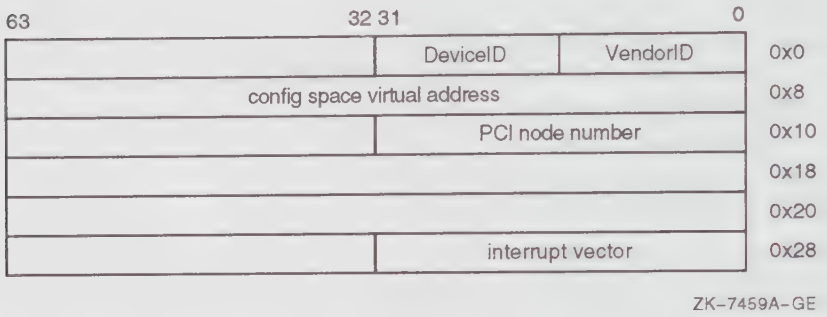
The PCI bus support module contains a PCI bus probe routine that steps through the device number of each potential PCI device on the host PCI and on any remote PCIs. For each PCI slot on the host PCI, the PCI probe routine attempts to read the vendor ID and device ID from the configuration space header of the device. If a device responds with a valid Vendor ID, the following information is recorded in the bus array entry for the PCI device:

Bus Array Entry Fields	Description
`busarray$q_hw_id`	Device ID in bits 31:16. Vendor ID in bits 15:0.
`busarray$q_csr`	Virtual address of base of PCI config space for this device.
`busarray$l_node_number`	PCI node number. Device number in bits 7:3. Function number in bits 2:0. Bus number in bits 15:8.

`busarray$l_int_vec`
`busarray$l_sys_irq`
`busarray$l_bus_specific_l`
	Interrupt SCB entry or system IRQ for this device.

Figure 14–4 shows a bus array entry for a PCI device found during bus probing.

Figure 14–4 Bus Array Entry for PCI Device During Bus Probing



14.6 Accessing Registers on PCI Buses

To access registers on a PCI device, you must do the following:

1. Determine the PCI physical address assigned to the device. (See Section 14.6.1.)
2. Map the PCI physical address into the processor's virtual address space using the IOC$MAP_IO routine. (See Section 14.6.2.)
3. Access the device using either the platform independent access routines IOC$READ_IO or IOC$WRITE_IO or using the CRAM data structure and associated routines IOC$CRAM_CMD and IOC$CRAM_IO.

14.6.1 Using IOC$READ_PCI_CONFIG and IOC$WRITE_PCI_CONFIG Routines

As described in Section 14.1.1, a PCI device may implement up to 6 base address registers in its configuration space header. On Alpha platforms, the console assigns PCI address space to each PCI device by writing a PCI physical address into a base address register.

OpenVMS Alpha provides two routines for accessing PCI configuration space. The formats of the IOC$READ_PCI_CONFIG and IOC$WRITE_PCI_CONFIG routines are as follows:

```
int ioc$read_pci_config (ADP *pci_adp, int pci_node, int offset, int length, int *data
int ioc$write_pci_config (ADP *pci_adp, int pci_node, int offset, int length, int data
```

Inputs:

`pci_adp`	Address of PCI ADP. Available to driver from IDB$PS_ADP.
`pci_node`	PCI node number. Function number in bits 2:0, device number in bits 7:3, bus number in bits 15:8. Available to driver from CRB$L_NODE. (driver must be loaded with /NODE qualifier).
`offset`	Byte offset in configuration space of field to be read or written.
`length`	Length of data field (expressed in bytes) to be read or written. Must be 1 (byte), 2 (word), 3 (tribyte), or 4 (longword).
`data`	For reads, a pointer to a longword cell to be written with the data read from configuration space. For writes, a longword containing the data to be written to configuration space.

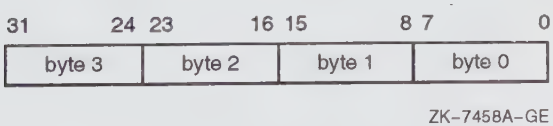
Outputs:

SS$_NORMAL	Success. For reads, data is returned in the caller's buffer. For writes, data is written to PCI configuration space.
SS$_BADPARAM	Failure. Could not find configuration space address for the specified PCI node number.

These routines acquire the MCHECK spinlock (raising IPL to 31) to ensure that they are single threaded. This is necessary because configuration space access involves manipulation of hardware registers in the host PCI interface. You must use these routines to access configuration space. Do not be tempted to use the configuration space base virtual address from the bus array.

These routines do not perform byte lane alignment of the data. For reads, data is returned in its natural byte lane. For writes, data must be positioned in its natural byte lane. In this context, natural byte lane means:

Figure 14–5 PCI Configuration Space Byte Lanes



For example, if you read a field of length 2 bytes from offset 2 in the configuration space header, the data will be returned in bits 31:16.

You should use IOC$READ_PCI_CONFIG to read the PCI physical address from the base address register(s) in your device's configuration space. The device specification should indicate which base address registers are used by your device and should give you their offsets in the configuration space header for the device.

The following example shows a call to the IOC$READ_PCI_CONFIG routine that reads the Vendor ID from PCI configuration space:

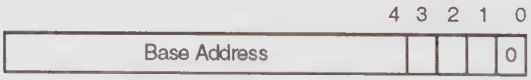
```
int     vendor_id;
int     status;

status = ioc$read_pci_config (pci_adp,
                              crb->crb$l_node,
                              0,    /* vendor id offset     */
                              2,    /* vendor id is 2 bytes */
                              &vendor_id);
```

The vendor ID will be returned in bits 15:0 of the vendor_id longword (the Vendor ID is a 2 byte field starting at configuration space offset 0).

14.6.1.1 PCI Configuration Space Base Address Register Format

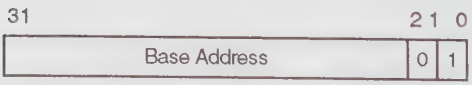
A PCI configuration space base address register can specify a PCI memory space address or a PCI I/O space address. The two forms of a base address register are as shown in Figure 14–6 and Figure 14–7.

Figure 14–6 Memory Format Base Address Register



ZK-7457A-GE

Figure 14–7 I/O Format Base Address Register



Bit 1 Reserved.
Bit 0 Set to one to indictate I/O format Base Address register.

ZK-7456A-GE

The specification for your device should state which base address registers are implemented and which PCI address spaces (memory or I/O) they describe. In general, if your device requires an address region in PCI I/O space, there will be a base address register that contains the starting address of the PCI I/O space assigned to your device. You can read this base address register using the IOC$READ_PCI_CONFIG routine. However, before you call the IOC$MAP_IO routine to map the PCI physical I/O address, you should clear bit 0 in the data returned from the read of an I/O format base address register before passing the address to the IOC$MAP_IO routine. For a PCI I/O address, you should specify the IOC$K_BUS_IO_BYTE_GRAN attribute in the call to IOC$MAP_IO.

Likewise, if your device requires an address region in PCI memory space, there will be a base address register that contains the starting address of the PCI memory space assigned to your device. You can read the base address register using the IOC$READ_PCI_CONFIG routine. You should clear bits 3:0 of the data returned from the read of a memory format base address register before passing the PCI physical address as an argument to the IOC$MAP_IO routine. For a PCI memory address, you can specify either the IOC$K_BUS_MEM_BYTE_GRAN attribute (to map device registers) or the IOC$K_BUS_MEM_DENSE attribute (for on-board device memory buffers). You should check the return status on calls to IOC$MAP_IO, because not all attributes are supported on all platforms.

If a call using one of the attributes fails, try the other attribute. If you cannot map the device, there is a problem and you should file a QAR.

If your device requires multiple address regions in PCI memory or I/O space, you should call IOC$READ_PCI_CONFIG and IOC$MAP_IO to map each region.

14.6.2 Mapping a PCI Physical Address

Once you have obtained the PCI physical address of a device, you must map the address into the processor's virtual address space.

The correspondence between PCI physical address and platform physical address varies according to whether you want the address to be mapped into PCI I/O space, PCI swizzled memory space, or PCI dense memory space. The platform physical address regions corresponding to each of the PCI address spaces differ from platform to platform. To abstract these differences, OpenVMS Alpha provides a platform independent I/O bus mapping routine called IOC$MAP_IO. IOC$MAP_IO allows a programmer to express a mapping request in terms of the device and desired access characteristics, without regard to the underlying platform address map and addressing techniques. For more information about using the IOC$MAP_IO routine, see Chapter 19.

The IOC$MAP_IO routine must be called at IPL 8 or lower because is may acquire the MMG spinlock; and the caller cannot be holding any spinlocks of higher rank than MMG.

In the following example, a call to the IOC$MAP_IO routine maps 64 KB of PCI I/O space in a region offering byte granularity starting at PCI I/O address 0.

```
int    status;
uint64 iohandle;
uint64 pci_physical_address = 0;

status = ioc$map_io (pci_adp,
                     crb->crb$l_node,
                     &pci_physical_address,
                     16*4096,
                     IOC$K_BUS_IO_BYTE_GRAN,
                     &iohandle);
```

14.7 Configuring a PCI Device and Loading A Device Driver

You can configure a PCI device and load a device driver manually using the System Management (SYSMAN) utility IO CONNECT command or by writing an IOGEN Configuration Building Module (ICBM) that is invoked by the SYSMAN IO AUTOCONFIGURE command. The following sections describe both methods.

14.7.1 Autoconfiguring a PCI Device

If you write an IOGEN Configuration Building Module (ICBM) to autoconfigure your device, the ICBM will be called when the upper level autoconfiguration routines find a PCI ADP. An ICBM performs the same steps described in this section and makes an explicit call to SYS$LOAD_DRIVER to configure your device. See Chapter 12 for more information about how to write an ICBM.

14.7.2 Configuring a PCI Device Manually

To configure a PCI device manually and load its driver, you can use the SYSMAN IO CONNECT command.

Manually configuring a PCI device using the SYSMAN IO CONNECT command is similar to configuring devices on other I/O buses. The differences are in specifying the /CSR qualifier and the /VECTOR qualifier.

To configure your device, use the following command to invoke the SYSMAN utility:

```
$ MCR SYSMAN
```

At the SYSMAN> prompt, enter the IO CONNECT command as follows:

```
$ SYSMAN> IO CONNECT devname
                    /driver = drivername
                    /vector = %x system_IRQ
                    /node = PCI_slot_number
                    /adapter = adapter_number
                    /csr = %x PCI_slot_base_virtual_address
```

devname
Specifies the OpenVMS device name of your device. This should be specified as a standard OpenVMS device name–a 2 letter device code, a controller letter, and a unit number.

adapter
This qualifier specifies the ADP that represents the bus to which your device is connected.

csr
Required by the driver loading program. The value specified in the /CSR qualifier is copied to the `idb$q_csr` field by the driver loading program. On some buses, this qualifier is used to tell the driver the physical address at which the device is located. However, for PCI, the physical address information is stored in the Configuration Space header of the device, as explained earlier. Therefore, this qualifier is not useful for PCI and should be specified as /CSR=0.

PCI device drivers are expected to call IOC$MAP_IO for address space mapping.

node
Identifies the PCI device to the PCI bus support routines. The value specified in the /NODE qualifier is copied to the `crb$l_node` by the driver loading program. Use the Node value from the IO SHOW BUS display that is associated with your device.

vector
Used by the driver loading program to connect your driver interrupt service routine to a hardware interrupt vector.

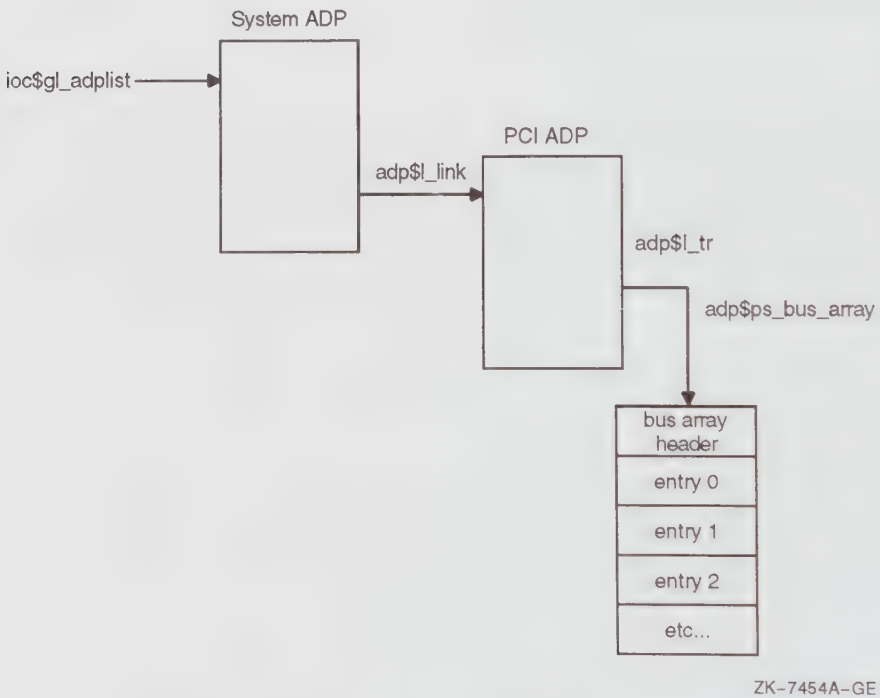
To find the interrupt vector for your device, you must run SDA on a running system. The ADP list and bus arrays are set up as shown in the following diagram. You must find the bus array entry for your PCI device. The bus array entry contains the interrupt vector offset that should be used as the value for the /VECTOR qualifier.

The ADP list on a platform with a PCI bus is as shown in Figure 14–8. Note that there may be intervening ADPs between the System ADP and the PCI ADP.

Note

There may be more than one PCI ADP. To determine the PCI bus to which your device is connected, you must use the SDA CLUE CONFIG command.

Figure 14–8 PCI Bus System ADP List



System global cell IOC$GL_ADPLIST points to a list of ADPs. An ADP is an OpenVMS Alpha data structure that represents an adapter. The PCI interface is an example of an adapter. The system ADP is always the first ADP in the list. Each ADP has a bus array, which is pointed to by ADP cell ADP$PS_BUS_ARRAY. A bus array consists of a header and a number of entries. There is an entry in the bus array for each device connected to the bus.

The structure definition of the ADP is available in [SYSLIB]SYS$LIB_C.TLB. You can use the Librarian utility to extract an ADP definition. For example:

```
LIBR /ALPHA /EXTRACT=ADPDEF /OUT=ADPDEF.H SYS$LIB_C.TLB
```

The structure definition of the bus array is also available in [SYSLIB]SYS$LIB_C.TLB. For example:

```
LIBR /ALPHA /EXTRACT=BUSARRAYDEF /OUT=BUSARRAYDEF.H SYS$LIB_C.TLB
```

14.7.3 Example

This example explains how to traverse the ADP list to find the interrupt vector offset for your PCI device.

- Invoke SDA and use the CLUE CONFIG command to find the TR number number associated with the display information of your device.
- Enter the following information:

  ```
  $ ANALYZE/SYS
  SDA> READ SYS$LOADABLE_IMAGES:SYSDEF
  SDA> FORMAT @IOC$GL_ADPLIST
  ```

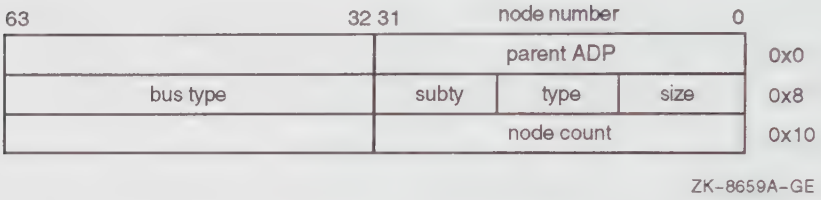
  SDA displays all of the fields of the system ADP.
- Find the address in the `adp$l_link` field.
- Format that address as follows:

  ```
  SDA> FORMAT ADDR_FROM_ADP$L_LINK
  ```

  SDA displays all of the fields of the next ADP in the ADP list. Search the `adp$l_link` pointers until you find the ADP with a TR number that matches the TR number in the `adp$l_tr` field. This matching TR number is the PCI ADP. Note that the PCI ADP will usually be the second ADP in the list.
- When you have found the PCI ADP, get the address from the `adp$ps_bus_array` field.

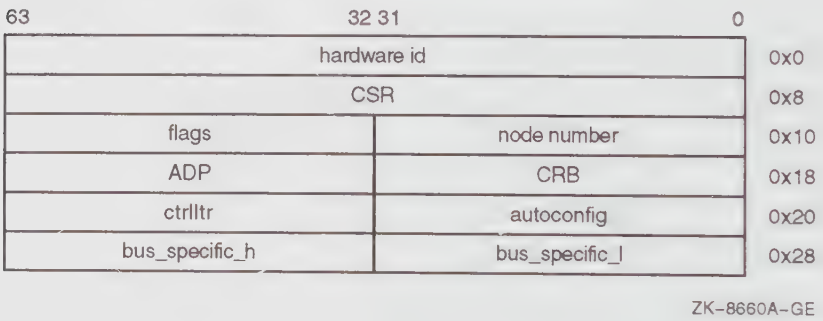
  Note that the SDA FORMAT command does not work on the bus array structure. The key points to remember about the bus array is that the header is three quadwords and each entry is BUSARRAYENTRY$K_LENGTH bytes. As of OpenVMS Alpha Version 7.0, BUSARRAYENTRY$K_LENGTH is seven quadwords. On releases prior to Version 7.0, BUSARRAYENTRY$K_LENGTH is six quadwords. Figure 14–9 shows the bus array header.

Figure 14–9 PCI Bus Array Header



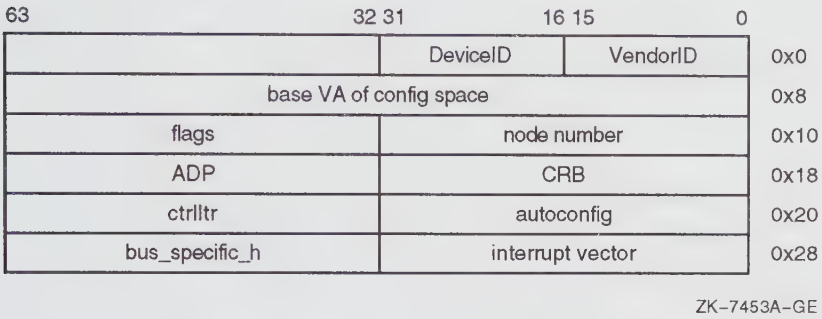
The bus array entries start after the bus array header. Figure 14–10 shows a generic bus array entry.

Figure 14–10 Generic bus array entry



A PCI bus array entry is shown in Figure 14–11.

Figure 14–11 PCI Bus Array Entry



ZK–7453A–GE

- When you have used SDA to find the PCI ADP and bus array, you should examine the bus array until you find the device ID and vendor ID of your device from PCI space. This information is the bus array entry for your device.

  Note that the interrupt vector offset is in the `bus_specific_l` field of the bus array entry for your device. This is the value that you should use for the /VECTOR qualifier in the SYSMAN IO CONNECT command.

  Once you have entered the interrupt vector value, it will not change when you reboot the system. However, if you move your device to a different slot or if you move your device to a different machine, you will have to find the new interrupt vector.

15

ISA Bus Support

Alpha platforms with Peripheral Component Interconnect (PCI) and Industry Standard Architecture (ISA) I/O bus support use industry standard bus components to implement the ISA bus interface. For this reason, the Alpha platforms provide the same set of bus resources for ISA options that are available on a PC. These bus resources are IRQ lines, DMA channels, I/O ports, and ISA memory buffers.

To allocate the bus resources to the various ISA devices that require them, some resource management is required. On ISA machines, the console provides an ISACFG command that allows a user to enter ISA configuration data into the system NVRAM. ISA bus support and bus configuration routines then extract the ISA configuration data from the console NVRAM. The console NVRAM ISA configuration data is referred to as the console **ISA configuration table**.

The console ISA configuration table contains resource usage information for each of the ISA devices in the system. An additional mechanism interprets the table entries and loads the correct drivers for the devices that are present. The OpenVMS **SYS$MANAGER:ISA_CONFIG.DAT** file stores driver and device name information. A user can edit this file to enter ISA bus configuration information as well as driver and device names. OpenVMS ISA bus configuration routines use the console ISA configuration table as a source of ISA bus resource usage and use the ISA_CONFIG.DAT file as a source of driver loading information, which includes driver and device names.

This chapter describes ISA device configuration on machines that support ISA as an I/O bus, and it contains procedures for configuring an ISA device. It also provides an example SYS$MANAGER:ISA_CONFIG.DAT file.

Note

For information about configuring ISA cards on an EISA bus, see Chapter 16.

15.1 OpenVMS ISA Bus Configuration

The basic strategy for ISA bus configuration in OpenVMS is to configure the devices found in the console ISA configuration table, and then configure the devices found in the ISA_CONFIG.DAT file.

During booting, the I/O bus mapping routine INI$IOMAP probes the console ISA configuration table and copies bus configuration information from the console ISA configuration table to the ISA ADP/Busarray. During autoconfiguration, IOGEN$ISA_CONFIG walks the ADP/Busarray and configures the devices found in the Busarray. IOGEN$ISA_CONFIG then opens file ISA_CONFIG.DAT and configures the devices found in the file.

15.2 Adding a Device

A machine fresh from the factory contains a console ISA configuration table with entries for all of the built-in devices (keyboard, mouse, com1, com2, lpt1, and floppy) and entries for any factory installed ISA options. Each console ISA configuration table entry lists the bus resources consumed by that device, as well as a 16 byte HANDLE (an ASCII character string), that is used to identify the device. OpenVMS will configure all factory installed and built-in devices using data from the console ISA configuration table. OpenVMS uses the HANDLE field of each console ISA configuration table entry to associate a device and a driver.

To configure an add-in ISA option device, you must first enter the IRQ used by the device in the console ISA configuration table as described in Section 15.2.1.

Note

This step is done prior to booting the operating system, so that the console knows that a particular IRQ is in use by an ISA device. This will prevent the console from assigning this IRQ to a PCI device. If you do not perform this step, the system might become unresponsive ("hang") because two devices are trying to use the same IRQ.

After you boot the operating system, you can load a driver for an ISA device in one of two ways:

- Using ISA_CONFIG.DAT file
- Manually by using the System Management (SYSMAN) utility

See Section 15.2.2 and Section 15.2.3 for more information about both of these methods.

15.2.1 Entering Interrupt Request Line (IRQ) Assignments

As mentioned in Section 15.2, before you add optional devices to an ISA bus, you must enter the IRQ used by the device in the console ISA configuration table using the ISACFG command as described in this section. For additional information about the ISACFG command, see the *AlphaStation 400 Series User Information* manual. The SRM console on your Alpha system assigns interrupt vectors to the PCI devices plugged into the PCI option slots. The console automatically assigns an available interrupt request line (IRQ) to each PCI device based on the configuration information stored by the console configuration utility with the system.

To avoid a conflict where an IRQ that you want to use for an ISA device is already reserved for a PCI device, use the console configuration utility to reserve the IRQ for the ISA device. As noted in Section 15.2, these conflicts might cause the system to become unresponsive because two devices are trying to use the same IRQ.

The commands you enter at the console prompt (>>>) are similar to the following:

```
>>> isacfg -mk -etyp 1 -enadev 1 -dev 0 -slot x -irq0 y
>>> init
```

Note the following conventions:

- *x* is the ISA option slot into which you insert the card.

  For more information about ISA option slots, see the NODE=x description in Section 15.2.2.

- *y* is the IRQ that you are reserving for your device.

Note the following as well:

- To verify that your command correctly reserved the IRQ, enter the following command:

  ```
  >>> isacfg -all
  ```

  The system will display the entire data structure of ISA configuration information stored in the console, including the IRQ you just reserved.

- If you made an error, you can enter the following command to cancel the assignment:

  ```
  >>> isacfg -rm -dev 0 -slot x
  ```

- If you are using your system as a Universal Platform booting the UNIX operating system, you may have already made an entry using the console configuration utility. You do not need to enter the information again. (If you do, the system will display a message informing you that the slot is already reserved for an existing ISA device.)

  If you want to override the existing information enter the following command:

  ```
  >>> isacfg -mod -slot x -dev 0 -etyp 1 -enadev 1 -irq0 y
  ```

  Note the following conventions:

  - *x* is the ISA option slot into which you insert the card.
  - *y* is the IRQ assigned to the device.

  Note that this specific command does not affect other configuration information values stored for the device; the command modifies only the IRQ information.

15.2.2 Configuring a Device with an ISA_CONFIG.DAT File

If you are using the ISA_CONFIG.DAT file to configure your device, you must first copy SYS$MANAGER:ISA_CONFIG.TEMPLATE to SYS$SPECIFIC:[SYSMGR]ISA_CONFIG.DAT.

Then edit the ISA_CONFIG.DAT file to specify the device name, driver name, and other ISA bus resource usage (such as DMA channels, I/O ports, and memory buffers) for your device. (This is analogous to editing config.sys on a PC.) You must enter the IRQ for your device in ISA_CONFIG.DAT, even though you already entered the IRQ in the console ISA configuration table (as described in Section 15.2.1).

The main advantage of using ISA_CONFIG.DAT for add-in ISA option configurations is that the device is automatically configured. There is no need to write an ICBM or to use the SYSMAN IO CONNECT command, even for devices that OpenVMS does not support.

The ISA_CONFIG.TEMPLATE file is shipped in the system manager's directory with examples of configuration entries for many ISA cards. The file is shipped with all entries commented out. An example entry from the SYS$MANAGER:ISA_CONFIG.TEMPLATE file is as follows:

```
;       Example 1: To indicate that the DE202 Card is plugged into slot 1,
;       using IRQ 5, I/O ports 300-30F, and memory buffer D0000-DFFFF, enter
;       the following into this SYS$SPECIFIC:[SYSMGR]ISA_CONFIG.DAT.
;
[ERA0]❶
NAME=ER❷                    ; device name
DRIVER=SYS$ERDRIVER❸        ; driver name
NODE=1❹                     ; plugged into ISA Option slot 1
IRQ=5❺                      ; device is using irq 5
PORT=(300:10)❻              ; 16 bytes starting at 300
MEM=(D0000:10000)❼          ; 64 Kbytes starting at D0000
USER_PARAM = thinwire❽      ; optional user parameter.  Available to
                                  ; driver via IOC$NODE_DATA call using function
                                    code of IOC$K_ISA_USER_PARAM=18.
```

❶ The opening bracket indicates that the following lines contain ISA configuration information. This is basically an entry delimiter. The text between the brackets is not used for anything.

❷ The NAME=xx field allows specification of a standard two letter OpenVMS device name. This two-letter string, along with a controller letter and a unit number, is passed to the driver loading service as a standard OpenVMS device name. The two-letter device string is also copied directly to the `busarray$q_hw_id` field in the ISA bus array entry specified by the `node` field.

❸ The DRIVER=drivername specifies the driver that should be loaded for this device. This name is limited to 16 ASCII characters, not including the implicit .EXE.

❹ The NODE=x field is used by OpenVMS to keep track of where ISA configuration information is stored in the operating system data structures. Note that any correspondence with the actual location of the device in a physical bus slot is not needed. However, if you are configuring more than one ISA device in the ISA_CONFIG.DAT file, you must use unique node numbers to avoid confusing the OpenVMS ISA bus configuration code. It might be helpful to label the ISA option slots 1, 2, 3, 4, etc. with a permanent felt-tip marker starting with the slot on the bottom (assuming the system is an upright model). Then use these slot numbers for the NODE=x portion of the ISA_CONFIG.DAT entries. When you enter the IRQ for your device at the console with the ISACFG command, the slot number that you specify must match the node number specified in the entry for the device in the ISA_CONFIG.DAT file.

❺ The IRQ=x field tells OpenVMS ISA configuration code which IRQ is being used by the device. OpenVMS ISA configuration code derives the interrupt vector that will be used by the driver interrupt service routine from the ISA IRQ.

Even though you specified the IRQ at the console with the ISACFG command, you also must specify the same IRQ here.

On AlphaStation 200 and 400 Series systems, the built-in devices use many of the available ISA IRQ lines as shown in Table 15–1.

Table 15–1 Available ISA IRQ Lines

IRQ	Device
0	TIMER (built-in)
1	KBD (built-in)
2	Dual 8259 cascade
3	COM2 (built-in)
4	COM1 (built-in)
5	Available for PCI or ISA option slot
6	FLOPPY (built-in)
7	LPT1 (built-in)
8	Interval timer (not used on these systems)
9	SOUND (built in device on AlphaStation 200 only), available for PCI or ISA option slot on AlphaStation 400
10	Available for PCI or ISA option slot
11	PCI NCR810 SCSI (built-in)
12	MOUSE (built-in)
13	DMA buffer chaining (not available outside PCI/ISA bridge)
14	Available for PCI or ISA option slot
15	Available for PCI or ISA option slot

Note that only IRQs 5, 9, 10, 14, and 15 are available for add-in devices on AlphaStation 400 Series systems. Only IRQs 5, 10, 14, and 15 are available on AlphaStation 200 Series systems because the built-in sound card uses IRQ 9. These are the only IRQ lines available for all of the PCI and ISA option slots.

You must be careful not to specify an IRQ that is in use by another device because there is currently no IRQ sharing in OpenVMS. Only one driver interrupt service routine can be connected to an IRQ. If you mistakenly overload an IRQ, only one driver (the one that is loaded last) will ever get interrupts from that IRQ.

❻ The PORT=(x:y) field allows specification of the I/O port usage of the option. Up to 9 I/O port ranges can be listed:

```
;   PORT=(aa:b,cc:d,...)     ; where aa and cc are the ISA I/O port base
;                            ; addresses for the device, and b,d indicate
;                            ; the number of bytes to be reserved.
;                            ; Up to 9 separate I/O port ranges can be
;                            ; specified.
```

If a driver needs to know which I/O ports its device is using, the driver can call the IOC$NODE_DATA routine to get the I/O port information. See Section 15.3 for more information about the IOC$NODE_DATA routine.

❼ The MEM=(x:y) field allows specification of the ISA memory buffers used by the ISA device. Up to 4 memory buffers can be specified:

```
;   MEM=(ee:f,gg:h,...)      ; where ee and gg indicate the base ISA memory
;                            ; address for the device's on-card buffer, and
;                            ; f,h indicate the size of the buffer in bytes.
;                            ; Up to 4 different memory buffer address
;                            ; ranges can be specified.
```

❽ The USER_PARAM= field is an optional user parameter that allows the user to create a string (72 bytes maximum) that is then available to the driver by way of the IOC$NODE_DATA routine. This technique is limited to applications that use the ISA_CONFIG.DAT file.

This particular example did not specifiy a DMA channel, because this ISA card does not use DMA channels. To specify a DMA channel, use the following:

```
DMA=(j,k,...)                ; where j and k are values (0-7) that
                             ; specify which channel(s) of the DMA
                             ; controller this device is using to
                             ; relay information.
```

You may specify up to 4 DMA channels for your device. Note that you can specify a DMA channel only on ISA buses, not on ISA devices on EISA buses.

After you have edited the ISA_CONFIG.DAT file, reboot the machine. Your device should be automatically configured, and the driver should be loaded.

15.2.3 Configuring an ISA Device Manually

To configure your ISA device manually , you can also use the System Management (SYSMAN) utility instead of using ISA_CONFIG.DAT file. The following example shows how to configure your device using the SYSMAN IO CONNECT command:

To configure your device, use the following command to invoke the SYSMAN utility:

```
$ MCR SYSMAN
```

At the SYSMAN> prompt, enter the IO CONNECT command as follows:

```
SYSMAN> IO CONNECT devname /ADAPTER=x /DRIVER=drivername -
/NODE=ISA_slot_number /CSR=%x ISA_slot_base_virtual_address /VECTOR=irq*4
```

The parameters to the IO CONNECT command are as follows:

devname
Specifies the OpenVMS device name of your device. This should be specified as a standard OpenVMS device name–a 2 letter device code, a controller letter, and a unit number.

/ADAPTER=x
Specifies the OpenVMS software adapter number of the ISA bus. In order to use the /ADAPTER argument with IO CONNECT command, you must indentify the TR number of the ISA ADP. To find this information, display the I/O adapters known to OpenVMS by entering the following command:

```
$ ANALYZE/SYSTEM
SDA> CLUE CONFIG
```

The CLUE CONFIG command displays system configuration and adapter configuration information. The TR number is located under the TR heading in the adapter configuration section. For example, the following display shows the output of the CLUE CONFIG command for an AlphaStation 400 Series system running OpenVMS V6.2:

```
SDA> CLUE CONFIG
```

```
Adapter Configuration:
----------------------
TR  Adapter    (Address)  Hose  Bus          Node  Device Name       HW-ID/SW
    Name
------------------------- ---- ----------------------------------------
1  KA0D02     (80D589C0)     0  BUSLESS_SYS
2  PCI        (80D58BC0)     0  PCI
                                    PKA:     6    NCR53C810         00011000
                                             7    SATURN            04848086
                                    EWA:     12   TULIP             00021011
                                    GQA:     13   PCI1280           30320E11
3  ISA          (80D5AC40)   0  ISA
                                             0    EISA_SYSTEM_BOAR  00000016
4  XBUS         (80D5B140)   0  XBUS
                                             0    MOUS              53554F4D
                                             1    KBD               0044424B
                                    TTA:     2    COM2              324D4F43
                                    TTB:     3    NS16450           00016450
                                    LRA:     4    LPT1              3154504C
                                    DVA:     5    AHA1742A_FLOPPY   504F4C46
```

In this display, you can see that the ISA bus adapter is TR number 3. Because you are plugging a device in to the ISA bus, specify /ADAPTER=3 in the SYSMAN IO CONNECT command. This will cause your driver data structures to be associated with the ISA bus data structures.

/DRIVER=drivername

The DRIVER=drivername qualifier specifies the file name of the driver that is to be loaded.

/NODE=slot

The /NODE=slot qualifier identifies the OpenVMS ISA bus array entry that will be used to store configuration information about your ISA device. Note that this need not have any correspondence with the physical ISA slot cutouts on the back of the machine. It is necessary, however, that you do not configure more than one device into the same slot. It is also necessary that the slot argument match the slot number that you specified when you entered the console ISACFG command to reserve the IRQ. As mentioned earlier, it is suggested that you label the ISA bus slots and use those slot numbers in the console ISACFG command, in the ISA_CONFIG.DAT file (if you decide to configure your device using this file), and in the /NODE=slot qualifier in the SYSMAN IO CONNECT command (if you decide to configure your device this way).

/CSR=baseaddress

The driver loading service that is called by the SYSMAN IO CONNECT command requires the /CSR=baseaddress input qualifier. The driver loading service copies the base address from the /CSR=baseaddress qualifier to the `idb$q_csr` field in the IDB. The Base CSR can be taken from the ISA ADP's `adp$q_csr` field, as shown in the following example:

```
SDA> READ SYS$LOADABLE_IMAGES:SYSDEF.STB

%SDA-I-READSYM, 6611 symbols read from SYS$COMMON:[SYS$LDR]SYSDEF.STB;1
SDA> FORMAT 80D5AC40

80D5AC40   ADP$Q_CSR                          828C8000
    80D5AC44                                      FFFFFFFF
    80D5AC48   ADP$W_SIZE                            0140
    80D5AC4A   ADP$B_TYPE                           01
    .
    .
    .
    .
    .
```

Note that `adp$q_csr` is a 64 bit field : FFFFFFFF.828C80000. The base address of ISA I/O space is a useful value for ISA device drivers and should be the value specified in the /CSR=baseaddress argument.

Note that if you code your driver using the IOC$MAP_IO routine to map your device register space, you may not need the baseaddress from the /CSR=baseaddress qualifier. However, you still must supply something for the /CSR=baseaddress qualifier, because it is required by the driver loading service. If you are using IOC$MAP_IO and you are not depending on any base address value being passed to you by the driver loading service, you should say /CSR=0 for this argument.

/VECTOR=irq*4

The /VECTOR=irq*4 qualifier specifies to the driver loading service the software interrupt vector in use by your device. Your driver interrupt service routine is connected to the software interrupt vector and will be invoked when your device generates a hardware interrupt. The irq value in this qualifier must be the same as the IRQ value that you specified when you entered the console ISACFG command to reserve the ISA irq.

15.3 Using IOC$NODE_DATA and IOC$NODE_FUNCTION Routines for ISA Buses

This section describes how to use the IOC$NODE_DATA and IOC$NODE_FUNCTION routines for a device driver using an ISA bus.

The bus support routine IOC$NODE_DATA is called by a driver to get bus slot/platform-specific information. The uses and capabilities of this routine vary from bus to bus. The bus support routine IOC$NODE_FUNCTION is called to perform bus slot-specific actions on behalf of a driver, such as enabling interrupts for a bus slot.

This section contains the prototypes for the IOC$NODE_DATA and IOC$NODE_FUNCTION routines. For more information about these routines, see Chapter 19.

The formats of the IOC$NODE_DATA and IOC$NODE_FUNCTION routines are:

```
int ioc$node_data (CRB  *crb, int  function_code, void *user_buffer)
int ioc$node_function (CRB  *crb, int  function_code)
```

Both of these routines use the `crb$l_node` field from the CRB to find the ADP and ISA bus array entry for the device. The supported function codes for IOC$NODE_DATA on the ISA bus are as follows:

IOC$K_EISA_IRQ	Return IRQ used by device
IOC$K_EISA_DMA_CHAN	Return DMA channel used by device
IOC$K_EISA_MEM_CONFIG	Return memory buffer(s) used by device
IOC$K_EISA_IO_PORT	Return I/O port(s) used by device

On the ISA bus, IOC$NODE_DATA uses the `crb$l_node` field from the CRB to find the ADP and the ISA bus array entry for the device. The ISA bus array entry contains a pointer to a data structure called an ISA_CFG_DATA (defined by `isacfgdef.h` in SYS$LIB_C.TLB) to find the requested information. The contents of the ISA_CFG_DATA are derived from two places. If you are configuring your device using the ISA_CONFIG.DAT file, the information returned by IOC$NODE_DATA is derived from the entry that you made in ISA_CONFIG.DAT for your device. If you are configuring your device using the SYSMAN IO CONNECT command, there should not be an entry in the ISA_CONFIG.DAT file. In this case the information returned by IOC$NODE_DATA is derived from the entry for your device in the console ISA configuration table, which you created when you used the console ISACFG command to reserve the IRQ for your device.

When calling IOC$NODE_DATA, you must supply a user buffer large enough to contain the information. The data size returned for each of the previously mentioned function codes is as follows:

IOC$K_EISA_IRQ	The IRQ used by the device is returned as a 32 bit int.
IOC$K_EISA_DMA_CHAN	The DMA channel number assigned to the device is returned as a 32 bit int. If more than one DMA channel is specified in ISA_CONFIG.DAT, then each one is returned as an int (an array of ints should be declared to receive the information).
IOC$K_EISA_MEM_CONFIG	The memory buffer used by the device is returned in an array of two consecutive 32 bit ints. The first array element contains the ISA bus base memory buffer address, and the second array element contains the memory buffer size (in bytes). If more than one memory buffer is specified in ISA_CONFIG.DAT, each one is returned in an array of ints. You should declare an array of ints that is big enough to contain all of the memory buffer information used by your device.
IOC$K_EISA_IO_PORT	The I/O port information used by the device is returned as 32 bit integers. The low 16 bits of the 32 bit integer contain the base I/O port, and the upper 16 bits contain the number of consecutive I/O ports from the base I/O port. If more than one I/O port is specified in ISA_CONFIG.DAT, each one is returned as a 32 bit int. You should declare an array of ints large enough to contain all of the I/O port information used by your device.

The supported function codes for IOC$NODE_FUNCTION on the ISA bus as follows:

IOC$K_ENABLE_INTR	Enables hardware device interrupt in the ISA interrupt control logic
IOC$K_DISABLE_INTR	Disables hardware device interrupt in the ISA interrupt control logic

IOC$NODE_FUNCTION also uses the `crb$l_node` field from the CRB to find the ADP and ISA bus array entry for the device. IOC$NODE_FUNCTION then finds the IRQ used by the device and either enables (if called with the IOC$K_ENABLE_INTR function code) or disables (if called with the IOC$K_DISABLE_INTR function code) the IRQ line used by the device.

15.4 Determining an Available ISA IRQ

To determine which IRQs are available for use by an ISA device, you must check a few things. The console utility will have IRQs reserved for system board devices, for ISA Option slot devices that have been entered by the user, and for any PCI Options that have been installed.

The file SYS$MANAGER:ISA_CONFIG.DAT may contain resource information for ISA option devices that the console does not know about.

In an AlphaStation 200 Series, the only available IRQs for ISA and PCI Option devices are 5, 10, 14, 15. In the AlphaStation 400 Series, the Audio option is not present, so the available IRQ list is 5, 9, 10, 14, 15. Note that thare are more option slots available than there are IRQs.

To determine if any of these IRQs are already consumed by devices previously installed in the system (either at the factory or by the user), you can follow this procedure:

1. Issue the console command ISACFG -ALL. This will output a list of all the ISA devices (and their resources) that the console knows about, including system board devices and any option slot devices that the user has entered. Search the output from this command to see if any of the IRQs from the available list are consumed by previously installed devices.

2. PCI Option slot devices report their interrupts through ISA IRQs also. The console is responsible for assigning these IRQs. It does so in a linear fashion, starting with the lowest numbered IRQ that is not specified as reserved in the console ISA configuration table. Each consecutive PCI slot is assigned the lowest available IRQ until all PCI options are assigned or we run out of IRQs. If there are no ISA Option slot device entries in the console ISA configuration table, the PCI Options would be assigned IRQs in order from the available list. If there is an entry for an ISA option slot device in the table, specifying IRQ 9, then any PCI Option slot devices would be assigned IRQs from the list 5,10,14,15. (Audio is a good example of this.)

   Use the console SHOW CONFIG command to list all the PCI devices. Note that the Intel SIO Bridge does not require an interrupt, and that the NCR 53C810 SCSI chip is hardwired to IRQ 11. Any other PCI devices that show up will consume IRQs from the available list in the manner previously described.

3. Boot the system and look through the file SYS$MANAGER:ISA_CONFIG.DAT for valid entries (not commented out). They will be specifying an IRQ. (Note that there should already be an entry reserving this IRQ in the console ISA configuration table if proper procedure is followed). If you find an IRQ listed that is not already specified in the console data, make an entry using the console ISACFG command as described earlier in this chapter.

   Any remaining IRQs are available for use by a new ISA Option card. Please note that you must reserve the IRQ using the console to ensure that an IRQ is reserved for your device.

15.5 Troubleshooting

This section contains information that might help you to solve some common problems that you might encounter while configuring your ISA device. If you have trouble adding a new device to the system, you can check a few things. More than likely, the device you are trying to add is conflicting with an existing device's ISA Resource assignments. For possible resource conflicts, check the output from the console's ISACFG -ALL command and compare with the contents of your SYS$MANAGER:ISA_CONFIG.DAT file.

Possible types of conflicts and suggested solutions include the following:

ISA IRQ conflicts
As mentioned in Section 15.4, on AlphaStation 200 and 400 Series systems there are more option slots than there are IRQs.

ISA I/O port, DMA channel, and memory buffer conflicts
Choose a new resource for your board, or adjust the existing boards resource so that the needed one is available.

Slot numbering conflicts
ISA does not have slot numbers. OpenVMS requires some way to keep track of devices in the ISA bus and uses slot number as a convenient method. If there are slot number conflicts in the ISA_CONFIG.DAT file, the OpenVMS code will overwrite the first driver's resource information with the last driver's information and fail to load the first driver.

You must also ensure that there are no slot number conflicts between the console data and the ISA_CONFIG.DAT data.

Valid ISA slot numbers for the AlphaStation Series 200/400 are 1, 2, 3, and 4.

If you made changes to your SYS$MANAGER:ISA_CONFIG.DAT file and can no longer boot, you should boot conversationally using the following command. (Remember to re-enable autoconfiguration after the problem has been resolved.)

```
>>> boot -flags 0,1 <boot_device>

SYSBOOT> SET NOAUTOCONFIGURATION 1
SYSBOOT> CONTINUE
```

This will boot OpenVMS without configuring any of your I/O devices. You can then edit the SYS$MANAGER:ISA_CONFIG.DAT file to correct any problems.

If you made changes to the console data and are prevented from booting, use the console ISACFG command to remove the changes that you made.

15.6 System Board Resources for AlphaStation 200 and 400 Series Systems

System board resources for AlphaStation 200 and 400 Series systems are as follows:

COM1:	IRQ=4, I/O port=3F8:8
COM2:	IRQ=3, I/O port=2F8:8
LPT1:	IRQ=7, I/O Port=378:8
FDC:	IRQ=6, I/O port=3F0:8, DMA chan=2
Mouse:	IRQ=C, I/O port=60,64
Keyboard:	IRQ=1, I/O port=60,64
TOY Clock:	I/O port=70,71
PCI NCR810 SCSI:	IRQ=B, I/O port=26,27
Sound:	(This is a system board device on AlphaStation 200 Series only) IRQ=9, I/O Ports=388:4,530:8, DMA chan=(0,1)
Timer /Counter:	IRQ=0

INTEL SIO PCI /ISA Bridge interrupt logic:	IRQ=2 (Cascade IRQ for the dual 8259 on the bridge chip) IRQ=D (used for DMA Buffer Chaining, unconnnected on these systems) IRQ=8 (used for interval timer, unconnected on these systems) DMA=4 (used by the DMA controller as the cascade line)

15.7 Sample ISA_CONFIG.DAT File

This section contains a sample ISA_CONFIG.DAT file that is similar to the file you can edit.

```
;
;       I S A _ C O N F I G . D A T
;
;       This file informs the OpenVMS AXP operating system which
;       devices are connected to the ISA bus.
;
;       Note: Do NOT make changes in this file.
;       To add option devices to this file you must first copy
;       this template file (SYS$MANAGER:ISA_CONFIG.TEMPLATE)
;       to SYS$SYSROOT:[SYSMGR]ISA_CONFIG.DAT.  Then edit the file
;       SYS$SYSROOT:[SYSMGR]ISA_CONFIG.DAT to add devices.
;
;   Contents of this file:
;   ----------------------
;       o Description of configuration command sets
;
;       o Example configuration command sets for supported ISA option cards
;
;       o System board resources for AlphaStation 200 and 400 series systems
;
;
;
;   Conventions used in this file:
;   ------------------------------
;       o Characters following a semi-colon (;) are comments.
;
;       o All numbers must be specified in hexadecimal.
;
;       o You must separate records in the file by using square brackets
;         around each ISA device that is specified (for example, [xyzn]).
;
;       o Spaces are ignored.
;
;       o Note the following about keywords:
;
;         - Keywords can be in any order, *except* in the following
;           instance: the NAME keyword must precede the NODE keyword or
```

```
;           the SYSMAN IO SHOW BUS command will not include the ISA
;           devices in the display.
;
;         - Each keyword must be on its own line.
;
;         - All keywords required for a device must be included before
;           the next record [xyzn] begins.
;
;    Description of Configuration Command Sets:
;    -------------------------------------------
;
;        Specify the following information for each ISA device. Note
;        that the NAME, DRIVER, NODE, and IRQ fields are required.
;
;        Note that fields that are not used must be omitted. For example, if
;        the device does not use a DMA channel you must omit the DMA parameter.
;
;        NAME=xx                  ; where xx is the 2-letter device code,
;                                 ; (for example: ER for the DE202)
;
;        DRIVER=driver_name       ; The name of the driver stored in
;                                 ; SYS$LOADABLE_IMAGES.
;                                 ; For example: SYS$ERDRIVER for the DE202, and
;                                 ; SYS$IRDRIVER for the Proteon Token Ring card.
;
;        IRQ=i                    ; where i is a value (0-F) that specifies which
;                                 ; ISA IRQ the device uses to report interrupts.
;
;        NODE=n                   ; where n is the ISA option slot number
;                                 ; the device is plugged into.
;
;        DMA=(j,k,...)            ; where j and k are values (0-7) that
;                                 ; specify which channel(s) of the DMA
;                                 ; controller this device is using to
;                                 ; relay information.
;                                 ;
;                                 ; Note the comma between indicators and the
;                                 ; parenthesis that indicates a list
;                                 ; of parameters.
;                                 ;
;                                 ; Up to 4 DMA channels can be specified.
;
;        PORT=(aa:b,cc:d,...)     ; where aa and cc are the ISA I/O port base
;                                 ; addresses for the device, and b,d indicate
;                                 ; the number of bytes to be reserved.
;                                 ; Up to 9 separate I/O port ranges can be
;                                 ; specified.
;
;        MEM=(ee:f,gg:h,...)      ; where ee and gg indicate the base ISA memory
;                                 ; address for the device's on-card buffer, and
;                                 ; f,h indicate the size of the buffer in bytes.
;                                 ; Up to 4 different memory buffer address
```

```
;                                   ; ranges can be specified.
;
;       FLAGS = n                   ; Currently, bits 1 & 2 are the only flags in
;                                   ; use.  Bit 1 indicates that the device being
;                                   ; configured is a SCSI adapter. Therefore, if
;                                   ; the adapter is a SCSI  device, set this bit
;                                   ; to 1. This will allow IOGEN$SCSI_CONFIG (which
;                                   ; probes for devices on the SCSI bus) to locate
;                                   ; the devices.
;                                   ;
;                                   ; Bit 2 indicates that the device should be
;                                   ; loaded with novector specified (No interrupt
;                                   ; is required for the device.  Set bit 2 if
;                                   ; you want your device loaded with the NOVECTOR
;                                   ; parameter
;
;       USER_PARAM = text           ; This parameter is dedicated solely for
;                                   ; driver writer use.  A pointer to a copy of
;                                   ; text is passed back in the user_buffer
;                                   ; parameter from a call to IOC$NODE_DATA
;                                   ; using a function code of IOC$K_ISA_USER_PARAM
;                                   ; (note that the text is limited to 72 chars)
;
;
;       ************************** IMPORTANT ********************************
;
;       o For all of the following examples, be sure to verify that the
;         switches and jumpers on the card being inserted are set to
;         match the specified parameters BEFORE you remove the comment
;         characters (;) from this file.
;
;       o Be sure that there are no conflicting resource assignments between
;         devices (including the built-in devices on the system board and the
;         PCI option slot device IRQ assignments.)
;
;       o Be sure to copy SYS$MANAGER:ISA_CONFIG.TEMPLATE to
;         SYS$SYSROOT:[SYSMGR]ISA_CONFIG.DAT before making changes.
;
;         A list of resources assigned to system boards follows the examples.
;
;       ***********************************************************************
;
;
;   Example Configuration Command Sets for Supported ISA Option Cards
;   -----------------------------------------------------------------
;
;       Following are examples of configuration command sets necessary to
;       automatically configure option cards that are not shipped with
;       the system.
;
;       Example 1: To indicate that the DE202 Card is plugged into slot 1,
```

```
;       using IRQ 5, I/O ports 300-30F, and memory buffer D0000-DFFFF, enter
;       the following into this file:
;
; [ERA0]
; NAME=ER
; DRIVER=SYS$ERDRIVER
;       NODE=1                  ; plugged into ISA Option slot 1
; IRQ=5
; PORT=(300:f)            ; 15 bytes starting at 300
; MEM=(D0000:10000)       ; 64Kbytes starting at D0000
;       USER_PARAM = thinwire   ; optional user parameter is passed directly
;                               ; back to driver
;
;       Example 2: To configure the DigiBoard PC/8 card, remove the comment
;       characters (;)  from the following. (Be sure to verify that the
;       switches are set to match the indicated resources, or change the
;       resource lists to match the switches.)  Note that the Master Status
;       register address is expected to be listed first by SYS$YSDRIVER.
;
; [TTA0]
; NAME=TT
; DRIVER=SYS$YSDRIVER
; IRQ=5
;       NODE=2                  ; plugged into ISA Option slot 2
; PORT=(390:8,398:8,3a0:8,3a8:8,3b0:8,3b8:8,3c0:8,3c8:8,3d0:8)
;
;       Note that this configures only a single port on the card.  The
;       remaining 7 ports each need to be connected manually using the
;       following command:
;
;       $ MCR SYSMAN IO CONNECT/NOADAP/DRIVER=SYS$YSDRIVER TTAx:
;       ; where x is the unit number
;
;
;       Example 3: To configure the Proteon Token Ring card using
;       IRQ A, DMA channel 7, I/O Ports A20-A3F, STP media and
;       speed of 16 Mbits remove the comment characters from the
;       following commands.
;
;       [IRA0]
;        NAME=IR
;        NODE=4
;        DRIVER=SYS$IRDRIVER
;        IRQ=A
;        DMA=7
;        PORT=(A20:20)
;        USER_PARAM="STP,16"
;
;
;   System Board Resources for AlphaStation 200 and 400 Series Systems
;   ------------------------------------------------------------------
;
```

```
;    COM1:
;        IRQ=4, I/O port=3F8:8
;
;    COM2:
;        IRQ=3, I/O port=2F8:8
;
;    LPT1:
;        IRQ=7, I/O Port=378:8
;
;    FDC:
;        IRQ=6, I/O port=3F0:8, DMA chan=2
;
;    Mouse:
;        IRQ=C, I/O port=60,64
;
;    Keyboard:
;        IRQ=1, I/O port=60,64
;
;    TOY Clock:
;        I/O port=70,71
;
;    PCI NCR810 SCSI:
;        IRQ=B, I/O port=26,27
;
;
;    Sound: (This is a system board device on AlphaStation 200 Series only)
;        IRQ=9, I/O Ports=388:4,530:8, DMA chan=(0,1)
;
;    Timer/Counter:
;        IRQ=0
;
;    INTEL SIO PCI/ISA Bridge interrupt logic:
;        IRQ=2 (Cascade IRQ for the dual 8259 on the bridge chip)
;        IRQ=D (used for DMA Buffer Chaining, unconnnected on these systems)
;        IRQ=8 (used for interval timer, unconnected on these systems)
;        DMA=4 (used by the DMA controller as the cascade line)
;
```

16

EISA Bus Support

The EISA bus, which is an extension of the ISA bus, is designed to allow ISA customers to protect their hardware investment. EISA increases the data path to 32 bits, adds software readable product IDs (essential to automatic configuration), contains slot-specific I/O addressing, and provides active-low level sensitive interrupts (potentially allowing true sharing of IRQ levels). It accepts 8 and 16 bit ISA cards and 32 bit EISA cards.

EISA offers higher bandwidth than ISA, due to the increased data path width, and is thus suited to faster processors. The EISA bus functionality is a superset of the ISA bus; cards that work on the ISA bus also work on the EISA bus.

On Digital systems, the EISA bus is used as an I/O bus, normally accessed through a PCI-to-EISA bridge, rather than as a system bus. This is transparent to the device driver.

This chapter describes Extended Industry Standard Architecture (EISA) bus concepts and implementations on OpenVMS Alpha platforms.

16.1 EISA Bus Resources

The EISA bus defines a limited set of resources shared across the system. Each device or adapter is designed to use some subset of the available EISA resources:

- Interrupt Request Levels (IRQs)
- DMA Channels
- I/O Port Addresses
- EISA Memory space addresses

An EISA bus supports a maximum of 15 device or adapter slots, but the number actually available on a given hardware platform is a function of the individual platform design. A device or adapter can typically operate correctly using a subset of each of the resource types. Because EISA resources are limited and EISA device vendors are many, the configuration of EISA-based systems can be a complicated task. Resources must be assigned in a conflict-free manner, allowing all devices to work properly. The assignment of these resources to devices is done by the **EISA Configuration Utility (ECU)**, which saves the system configuration information and resource assignment in a database that is later available to the operating system. The following sections briefly describe each of these resources and provide a description of the ECU.

16.1.1 IRQs

The EISA specification defines 16 IRQ levels: 0 to 15. These wires are routed to an interrupt controller, which performs priority resolution and notification of the CPU. Note that some system platforms reserve certain IRQ levels for integrated devices. EISA cards typically have a software programmable register that is set to inform the hardware which IRQ line to use. ISA cards are not programmable; they have jumpers on the card which must be set to tell the hardware which IRQ line to use. An IRQ level is assigned to a card by the EISA Configuration Utility, which is described in Section 16.1.5.

EISA IRQ lines can be programmed in two modes; edge-triggered (active high) or level-sensitive (active low for sharing IRQs). Digital systems use level-sensitive settings.

Because Digital systems use the EISA bus as an I/O bus rather than as a system bus, the EISA interrupt mechanism becomes a subset of the overall system interrupt mechanism. The EISA IRQ (that is, the line that the card is programmed to drive) is not always the same as the "system IRQ", or "vector". Section 16.6 describes system and EISA IRQs.

OpenVMS Alpha versions through Version 7.0 allow one EISA IRQ per device.

16.1.2 DMA Channel

For cards that do not have Bus Master capability (that is, they are not complex enough to take control of the EISA bus), EISA provides seven DMA channels. The driver must set up the registers (base address, count, etc.) corresponding to the assigned DMA channel. For more information about DMA, see Section 16.3 . The ECU is responsible for assigning the DMA channels to cards that require them.

16.1.3 I/O Port Addresses

The EISA bus defines 64 KB of I/O port space. The first 4 KB is reserved for system board and ISA devices. The remaining I/O port space (4 to 64 KB) is EISA slot-specific address space with a 4 KB range reserved for each EISA slot. To avoid two cards responding to the same I/O address and to provide increased configuration flexibility, ISA cards provide a jumper that selects the range of I/O addresses in the low 4 KB to which the card responds.

The ECU is responsible for assigning I/O Port starting addresses to ISA cards without overlapping. For more information about the available ISA I/O ports, refer to the system address maps in Appendix A. EISA cards generally do not need an I/O port because they understand slot-based addressing.

16.1.4 EISA Memory Addresses

The EISA bus defines 46 KB of memory space. On some personal computers, the EISA bus is the system bus, which means that system memory and EISA devices share the 46 KB memory space. On Digital systems, EISA memory space is accessible through Alpha I/O address space, and the EISA bus is used as an I/O bus. Digital systems make EISA memory space available to access on-board memory, such as frame buffers, on EISA cards requiring such functionality.

16.1.5 EISA Configuration Utility

The ECU is a standalone program that runs without the operating system. It assigns resources to all cards on the EISA bus in a conflict-free manner. The ECU reads configuration files (.CFG files) to obtain resource requirements for the EISA devices in the system, including system board devices and prompts for user input to obtain resource requirements for the ISA devices in the system.

The EISA protocol incorporates a software readable product ID at a predefined offset in slot-specific EISA I/O port address space. The ID is stored as 4 bytes of compressed information, and decompressed into a 3-character manufacturers code followed by a 4-character card-specific identifier. The ECU determines which devices are present and uses their IDs to locate configuration files, which contain initialization information and resource requirements. It determines the requirements of all the options and finds a conflict-free assignment of system resources if possible. It writes all the configuration data to NVRAM, where it is available to system firmware for use during system initialization. The EISA specification provides details on the contents of configuration files.

The OpenVMS Alpha operating system makes ECU data available to device drivers using the IOC$NODE_DATA routine. A device driver calls this routine to determine which resources are assigned to its device.

Note that the ECU need only be run when the system configuration changes, that is, when boards are added, deleted, or moved in the EISA slots.

16.1.6 Resource Assignment on Digital Systems

Each Digital system supporting an EISA bus ships with an ECU configuration file that defines the EISA slots used by integrated devices (if any), the resources used by the integrated devices, and the slots available for additional cards.

16.2 EISA Interrupts

An EISA device driver uses the IOC$NODE_DATA routine to determine which IRQ has been selected for the device by the ECU. The driver then programs the device to use that IRQ, if necessary, and enables interrupts at both the system level (using the IOC$NODE_FUNCTION routine) and at the device level. For more information about the IOC$NODE_DATA and IOC$NODE_FUNCTION routines, see Section 16.5.

On OpenVMS Alpha, the device interrupt flow proceeds as follows:

1. The EISA device requires an interrupt, and asserts its programmed EISA IRQ line.
2. A platform-specific interrupt control mechanism notifies the CPU of the request.
3. PALcode is invoked. It determines that the interrupt is an EISA I/O interrupt, acknowledges the interrupt, receives a vector identifying the device, and uses that vector to locate the appropriate SCB vector. PALcode dispatches to the operating system through the SCB vector.
4. For an EISA bus, the SCB vector normally locates an operating system routine that dispatches to the device driver interrupt service routine (that is, EISA interrupts are indirectly vectored).
5. The device driver interrupt service routine services the interrupt and returns to the operating system.
6. The operating system issues an End Of Interrupt (EOI) command, which reenables interrupts of equal and lower priority. Note that if software does not perform the EOI, all future interrupts of equal or lower priority remain disabled.

16.3 EISA DMA Support

EISA systems provide seven independently programmable DMA channels for use by EISA/ISA cards that do not have Bus Master capability, and that cannot drive the necessary signals to perform DMA on their own. Each channel can be programmed for 8, 16, or 32 bit DMA device size, and ISA compatible, "type a", "type b", or burst dma "type c" modes. An EISA bus controller chip handles the data size translation. The DMA addressing circuitry supports full 32 bit addresses for DMA devices. Each channel includes a 16 bit Current register, an 8 bit Low Page register and an 8 bit High Page register, for the total 32 bits. The channels can be programmed for one of four different transfer modes: single, block, demand, and cascade. The DMA controller also offers buffer chaining, auto-initialization, and support for a ring buffer in memory. Buffer chaining is not supported in OpenVMS Alpha.

Note that while the DMA controller provides per-channel programmable features, such as transfer mode, a number of the DMA controller registers in which such features are programmed are shared among the channels. For instance, DMA channels 0 to 3 share a single write-only MODE register, and DMA channels 4 to 7 also share a single write-only MODE register. Access to such a register must be synchronized among the device drivers performing DMA.

Documentation that describes the 82357 DMA controller is available from INTEL.

16.4 EISA I/O Address Map

EISA defines a 16 bit I/O address composed of a four bit slot number in bits <15:12> and 12 bits, or 4 KB, of I/O space per EISA slot. However, as a side effect of keeping the EISA bus backwards compatible with the ISA bus, certain ranges of I/O space can be used only by ISA cards.

ISA uses a 10 bit I/O address, so an ISA device can only access addresses in the range 000 to 3FF. However, ISA reserves addresses 000 to 0FF for the ISA system board (slot 0). Therefore, normal ISA devices respond only to addresses in the range 100 to 3FF, that is, addresses in which bits <9:8> are non-zero.

The EISA specification therefore requires that EISA devices only respond to addresses in which bits <9:8> are zero, eliminating the possibility of interfering with ISA devices. The following list shows the useable I/O address range for an EISA device, where slot_number is a 4-bit value from 1 to F.

- Slot_number + 000 to slot_number + 0FF
- Slot_number + 400 to slot_number + 4FF

- Slot_number + 800 to slot_number + 8FF
- Slot_number + C00 to slot_number + CFF

ISA I/O address space is accessed as Slot 0, addresses 100 to 3FF. Since ISA devices decode only 10 I/O address bits, be careful of duplication. All of the following addresses map to the same range (0100 to 03FF) if only the low 10 bits are considered. Again, slot_number is any 4-bit value from 1 to F.

- 0500 to 07FF
- 0900 to 0BFF
- 0D00 to 0FFF
- Slot_number + 100 to slot_number + 3FF
- Slot_number + 500 to slot_number + 7FF
- Slot_number + 900 to slot_number + BFF
- Slot_number + D00 to slot_number + FFF

16.5 Using IOC$NODE_DATA and IOC$NODE_FUNCTION Routines for EISA Buses

The IOC$NODE_FUNCTION routine provides a platform-independent way to enable or disable EISA interrupts at the system level. The IOC$NODE_DATA routine returns information on the resources assigned to the device by the ECU. A device driver's controller or unit initialization routine normally would use IOC$NODE_DATA to determine which IRQ has been selected for the device by the ECU. The driver then programs the device to use that IRQ, if necessary, and enables interrupts at both the system level (using IOC$NODE_FUNCTION) and at the device level.

For device drivers written in C, function codes for IOC$NODE_FUNCTION and IOC$NODE_DATA are defined in `iocdef.h`, and function prototypes are in `ioc_routines.h`, both in SYS$LIB_C.TLB. The following sections briefly describe IOC$NODE_DATA and IOC$NODE_FUNCTION. For more information about these routines, see Chapter 19.

16.5.1 IOC$NODE_DATA

The IOC$NODE_DATA routine returns platform-specific data necessary for drivers. For an EISA device, it is used to obtain the resources assigned to the device by the ECU. The resources relevant to EISA devices are IRQ, DMA Channel number, EISA IO Port address, and EISA Memory address.

On OpenVMS Alpha versions through Version 7.0, the EISA device can be configured with one IRQ, a maximum of 4 DMA Channels, a maximum of 20 I/O Ports, and a maximum of 9 memory addresses. For devices configured with more than one of a given resource, IOC$NODE_DATA returns all values. It does not verify that the size of the driver-specified buffer is large enough; IOC$NODE_DATA assumes that it has been called with a buffer large enough to hold whatever resources it locates.

The format of IOC$NODE_DATA is:

```
int ioc$node_data (CRB  *crb, int  function_code, void *user_buffer)
```

The parameters to IOC$NODE_DATA are:

`crb`	Address of CRB. IOC$NODE_DATA obtains the EISA slot number from the CRB. For a manually connected driver, the slot number is the value specified in the "/NODE=x" qualifier.
`function_code`	Function to be performed. IOC$NODE_DATA accepts four function codes relevant to EISA devices; IOCK_EISA_IRQ, IOCK_EISA_DMA_CHAN, IOC$K_EISA_IO_PORT, and IOC$K_EISA_MEM. See Table 16–1 for more details about the function codes.
`user_buffer`	Address of a driver-supplied buffer to contain the returned data. IOC$NODE_DATA assumes that the buffer is large enough for all the returned data.

Table 16–1. IOC$NODE_DATA Function Codes for EISA Buses

IOC$K_EISA_IRQ	Return IRQ used by device. IOC$K_EISA_IRQ requires a longword buffer. It returns the single IRQ assigned to the device in bits <15:0> and indicates whether the ECU specified edge-triggered or level-sensitive interrupts in bit 16 (0=edge, 1=level).

(continued on next page)

Table 16–1 (Cont.) IOC$NODE_DATA Function Codes for EISA Buses

IOC$K_EISA_DMA_CHAN	Return DMA channel used by device. IOC$K_EISA_DMA_CHAN requires a longword buffer per DMA channel. For each DMA channel assigned to the device, it returns: • Channel number - bits <15:0> • Device transfer size - bits <18:19> 8 bit : 00 16 bit : 01 32 bit : 11 • Transfer timing information - bits <21:20> ISA compatible timing : 00 EISA type A : 01 EISA type B : 10 EISA type C : 11 (burst mode)
IOC$K_EISA_MEM_CONFIG	Return memory buffer(s) used by device. IOC$K_EISA_MEM requires a quadword buffer per Memory address. It returns the assigned starting address of EISA memory in the first longword, and the size in bytes in the second longword. These addresses are used to access on-board RAM. The returned starting address is an EISA bus physical address. To use this address, it must be converted to a platform-specific address using the IOC$MAP_IO routine, specifying the function code IOC$K_BUS_IO_BYTE_GRAN or IOC$K_BUS_DENSE_SPACE. IOC$MAP_IO returns an `iohandle`, which is used later as input to the IOC$READ_IO and IOC$WRITE_IO routines.

(continued on next page)

Table 16–1 (Cont.) IOC$NODE_DATA Function Codes for EISA Buses

IOC$K_EISA_IO_PORT	Return I/O port(s) used by device. IOC$K_EISA_IO_PORT requires a longword buffer per I/O Port. It returns the starting address of the assigned I/O port in bits <15:0> and the number of consecutive bytes assigned in bits <31:16>. The returned starting address is an EISA bus physical address. To use this address, it must be converted to a platform-specific sparse space address using the IOC$MAP_IO routine, specifying the function code IOC$K_BUS_IO_BYTE_GRAN. IOC$MAP_IO returns an `iohandle`, which is used later as input to the IOC$READ_IO and IOC$WRITE_IO routines.

16.5.2 IOC$NODE_FUNCTION

The IOC$NODE_FUNCTION routine provides a platform-independent way to enable or disable EISA interrupts at the system level. The formats of the IOC$NODE_FUNCTION routine is:

```
int ioc$node_function (CRB  *crb, int  function_code)
```

It accepts two parameters, the CRB address and the function code. Legal function codes are IOC$K_ENABLE_INTR and IOC$K_DISABLE_INTR; others return the error status SS$_ILLIOFUNC.

IOC$NODE_FUNCTION obtains the EISA slot number from the `crb$l_node` field in the CRB. It verifies that the slot is valid, obtains the device's IRQ, and relies on its platform-specific knowledge to enable or disable the IRQ in a manner appropriate for the system interrupt mechanism.

16.6 Configuring an EISA Device Manually

The OpenVMS Alpha operating system can only autoconfigure devices that it recognizes. For OpenVMS Alpha versions through Version 7.0, customer-written drivers need to be connected manually using the SYSMAN IO CONNECT command. Note that EISA resources are not specified on the IO CONNECT command. EISA resource information can only be specified by means of the ECU, which should be run before loading the driver.

To configure your device, use the following command to invoke the SYSMAN utility:

```
$ MCR SYSMAN
```

At the SYSMAN> prompt, enter the IO CONNECT command as follows:

```
$ SYSMAN> IO CONNECT devname
   /driver = drivername
   /vector = %x system_IRQ
   /node = EISA_slot_number
   /adapter = adapter_number
   /csr = %x EISA_slot_base_virtual_address
```

devname
Specifies the OpenVMS device name of your device. This should be specified as a standard OpenVMS device name–a 2 letter device code, a controller letter, and a unit number.

vector
For most Alpha systems:

```
/vector=(EISA IRQ * 4)
```

For AlphaServer 2100, 2100A, and 8200/8400 systems:

```
/vector=((EISA IRQ + 7) * 4)
```

Because Digital systems use the EISA bus as an I/O bus rather than as a system bus, the EISA interrupt mechanism is a subset of the overall system interrupt mechanism. The actual interrupt vector might differ from the limited range available to an EISA device. This is reflected in the two different formulas used by current Alpha systems, shown above.

EISA interrupts are indirectly dispatched because End-Of-Interrupt processing must be performed by the operating system to dismiss the interrupt. The device's SCB vector, therefore, locates an operating system dispatch routine rather than the device's interrupt service routine (ISR).

The vector parameter is a platform-specific offset based on the EISA IRQ. It is used by the operating system to locate the device driver's ISR in a table pointed to by ADP$L_VECTOR.

node

```
/node = EISA_slot_number
```

EISA_slot_number is a value from 1 to 15 (slot 0 is reserved for the system board). The console command SHOW CONFIG displays populated EISA slots and the EISA software ID of devices in the slots.

The DEC 2000 differs from other systems in that its node parameter must be a hexadecimal value containing the EISA IRQ in the high word and the EISA slot number in the low word.

adapter

```
/adapter= adapter_TR_number
```

Identifies the bus on which the device resides. The SDA command CLUE CONFIG displays the TR for the EISA bus.

CSR

```
/csr=  %x EISA_slot_base_virtual_address
```

EISA reserves slot 0 for the system board; slots 1 to 15 are available for devices. Each EISA slot has a reserved 4 KB I/O address space. The CSR is the virtual address of the slot's I/O address space. It is the base address used by the CRAM routines to access device registers. The CSR parameter is also stored in the idb$q_csr field.

During system initialization, the operating system maps a contiguous virtual address range sufficient for the maximum number of EISA slots. This is a sparse space mapping, so the actual amount of virtual address space per slot is a function of the I/O address swizzle factor for the system.

To compute the CSR assigned to a particular slot if the system swizzle is unknown, do the following:

1. Use the SDA command CLUE CONFIG to locate the EISA ADP structure.
2. Examine eisa adp + adp$ps_bus_array to locate the EISA busarray structure.
3. Skip past the busarray header.
4. Multiply the size of a busarray entry by the slot number to locate the busarray entry for your slot.
5. Examine busarray$q_csr.

To obtain the swizzle factor, use the IOC$NODE_DATA routine with the function code IOC$K_IO_ADDRESS_SWIZZLE. Existing systems use a swizzle factor of 5, except for the DEC 2000 which uses a swizzle factor of 7. However, future systems could use other values.

To compute the CSR assigned to a particular slot if the system swizzle is known, do the following:

1. Use the SDA command CLUE CONFIG to locate the EISA ADP structure.
2. Examine `eisa adp + adp$q_csr` to obtain the base address for EISA slot 0.
3. Multiply the slot number of interest by 4 KB.
4. Left-shift the resulting value by the swizzle factor.
5. Add to the value in the `adp$q_csr`.

Section 16.6.1 demonstrates both methods to obtain the CSR for EISA slot 3, on a system with a swizzle factor of 5. Note that all values are hexadecimal.

16.6.1 Example: Locating the CSR for an EISA Device

```
$ ana/sys
SDA> read sys$loadable_images:iodef
SDA> clue config
Adapter Configuration:
----------------------
TR  Adapter Name (Address)  Hose  Bus          Node  Device Name       HW-Id/SW
--  ----------------------  ----  -----------  ----  ----------------  --------
 1              (81133480)     0  CBUS
                                                0  KA0902_CPU        00000017
                                                5  KA0902_MEM        00000018
                                                8  KA0902_IIO        00000019
 2  PCI         (81133880)     0  PCI
                                                2  MERCURY           04828086
                                          GYA:  7  TGA               00041011
 3  EISA        (81133F00)     0  EISA
                                                0                    012AA310
                                          PAA:  2  KFESA             002EA310
                                          PAB:  3  KFESA             002EA310
 4  XBUS        (81134440)     0  XBUS
! First demonstrate the computation when the swizzle is known
SDA> ex 81133f00 + adp$q_csr       ! EISA ADP is 81133F00
81133F00:  FFFFFFFF 87AE0000       ! EISA ADP CSR is 87AE0000, slot 0 base VA
SDA> ev 1000 * 3                   ! Multiply 4 KB by 3 (for slot 3)
Hex = 00003000
SDA> ev 3000 @ 5                   ! Left shift by 5 (5 is swizzle factor)
Hex = 00060000
SDA> ev 87ae0000 + 60000           ! Add slot 3 offset to base. CSR = 87B40000
Hex = 87B40000
! Now demonstrate walking the busarray
```

```
SDA> ex 81133f00                          ! Locate the EISA ADP
81133F00:  FFFFFFFF 87AE0000

SDA> ex @(.+adp$ps_bus_array)             ! Locate the EISA busarray
81134100:  00000000 81133F00

SDA> ex .+busarrayheader$k_length         ! Skip the busarray header
81134118:  00000000 012AA310

SDA> ex .+(3*busarrayentry$k_length)      ! Locate slot 3 in busarray
811341A8:  00000000 002EA310

SDA> ex .+busarray$q_csr
811341B0:  FFFFFFFF 87B40000     ! CSR matches computed value
```

16.7 Configuring an ISA device in an EISA Slot

Because ISA devices do not have a software-readable ID, they cannot be detected by operating system software or by console firmware. Therefore, no information is displayed for EISA slots populated by ISA devices.

To create an ECU entry for an ISA device, a configuration file for the device must exist either on the ECU diskette or on another FAT-format diskette.

The ECU provided by Digital contains a generic ISA configuration file, ISA0000.CFG, that can serve as an example. This configuration file can be copied to another diskette and edited using a personal computer. The EISA specification provides details on configuration file syntax and format.

To create the ECU entry for an ISA device in an EISA slot:

1. Run the ECU from the console prompt.
2. From the ECU main menu, choose Step 2 (Add or remove board).
3. Select a slot for the ISA card and press enter.
4. Insert the diskette containing the configuration file for the ISA device and press enter.
5. Press F10 to exit to the main menu.
6. Choose Step 3 (View or edit details).
7. Examine the information for the slot and adjust as needed. To change a resource, position the cursor to that resource and press F6.
8. Press F10 to exit to the main menu.
9. Select Step 5 and exit, saving the changes.

After creating the ECU entry, adjust any necessary jumpers on the card to reflect the assigned IRQ, DMA channels, etc. Insert the card in the chosen slot.

To connect an ISA device in an EISA slot, execute the ECU as described in the previous procedure. Make a note of the slot number, because this information will be required for the /NODE input to SYSMAN. Compute the system_IRQ as described for EISA devices. Use the TR of the EISA bus for the /ADAPTER parameter.

All ISA devices use the same CSR. By convention, ISA I/O address space is accessed as EISA slot 0 (ISA devices ignore the 4-bit slot portion of the EISA I/O space address). Therefore, the CSR for all ISA devices is the base virtual address of EISA slot 0, the base of EISA I/O space. To obtain this value, use the SDA command CLUE CONFIG to locate the EISA ADP data structure. Format the EISA ADP and use the value in the `adp$q_csr` as the CSR parameter for all ISA cards.

16.7.1 Example: Locating the CSR for an ISA Device in an EISA Slot

```
$ ana/sys
SDA> read sys$loadable_images:iodef
SDA> clue config
Adapter Configuration:
----------------------
TR  Adapter Name (Address)  Hose  Bus          Node  Device Name       HW-Id/SW
--  ----------------------  ----  -----------  ----  ----------------  --------
 1              (81133480)     0  CBUS
                                                0  KA0902_CPU        00000017
                                                5  KA0902_MEM        00000018
                                                8  KA0902_IIO        00000019
 2  PCI         (81133880)     0  PCI
                                                2  MERCURY           04828086
                                          GYA:  7  TGA               00041011
 3  EISA        (81133F00)     0  EISA
                                                0                    012AA310
                                          PAA:  2  KFESA             002EA310
                                          PAB:  3  KFESA             002EA310
 4  XBUS        (81134440)     0  XBUS

SDA> ex 81133f00 + adp$q_csr          ! The EISA ADP CSR is the base for slot 0
81133F00:  FFFFFFFF 87AE0000          ! All ISA devices use CSR 87AE0000
```

Part V

Reference

Part V provides details about the data structures, entry point routines, system routines, and C macros used to create OpenVMS Alpha device drivers.

17

Data Structures

This chapter describes data structures referenced by driver code. While the list is not comprehensive, this chapter covers many of the data structures and fields that are most important to device-driver writers. Section 17.1 lists and briefly defines each type of structure included in this chapter. The sections that follow provide a more detailed description and lists their fields in the order in which they appear in the structures.

Header files for all of the data structures described in this chapter are available in SYS$LIBRARY:SYS$LIB_C.TLB. For example, the following files are for the IRP, PCB, and UCB data structures:

```
#include <irpdef.h>
#include <pcbdef.h>
#include <ucbdef.h>
```

All data structures discussed in this chapter, with the exception of the channel control block (CCB), exist in nonpaged system memory.

17.1 Overview of I/O Database Data Structures

Components of the I/O database include the following:

- Structures that describe individual hardware components, such as devices, controllers, and adapters. This category includes the following structures:

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)

Structure	Description	Associated Structures
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	—

Structure	Description	Associated Structures
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)

Structure	Description	Associated Structures
Controller register access mailbox (CRAM)	Describes a read or write transaction to device interface register space	—

17.2 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 17–1 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.

Table 17–1 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 routine 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.

(continued on next page)

Table 17–1 (Cont.) Contents of Adapter Control Block

Field	Use
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_C.TLB) into this field when the routine creates an ADP.

(continued on next page)

Table 17–1 (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 17.6) 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.

(continued on next page)

Table 17–1 (Cont.) Contents of Adapter Control Block

Field	Use
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. See the discussion at the beginning of Section 17.2, for an example of parent, child, and peer ADP relationships.
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. See the discussion at the beginning of Section 17.2, for an example of parent, child, and peer ADP relationships.
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. See the discussion at the beginning of Section 17.2, for an example of parent, child, and peer ADP relationships.
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.

(continued on next page)

Table 17–1 (Cont.) Contents of Adapter Control Block

Field	Use
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 17–1 (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.	

(continued on next page)

Table 17–1 (Cont.) Contents of Adapter Control Block

Field	Use
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.
adp$l_intd	Interrupt transfer vector. For adapters that service indirect interrupts (`adp$v_indirect_vector` in `adp$l_adapter_flags` is set), this 4-longword field (described in Section 17.6) provides information used by OpenVMS Alpha to service a device interrupt, such as the location of the ADP and its indirect interrupt service routine. See the description of the `adp$l_vector` field for additional information on how the adapter services indirect interrupts.

17.2.1 BUSARRAY (Bus Array)

The bus array data structure (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 17–2 describes the fields of the BUSARRAY structure; Table 17–3 describes the contents of each entry in the bus array.

Table 17–2 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.

(continued on next page)

Table 17–2 (Cont.) Contents of Bus Array

Field	Use	
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$l_bus_type	Type of tightly coupled I/O interconnect or multichannel I/O widget the BUSARRAY describes. The adapter initialization routine writes this field when it creates the BUSARRAY. The following constants represent the interconnects supported on OpenVMS Alpha systems:	
	BUS$_FBUS	Futurebus
	BUS$_XMI	XMI
	BUS$_LBUS	DEC 4000 Alpha LBUS
	BUS$_TURBO	TURBOchannel
	BUS$_CBUS	DEC 4000 Alphasystem 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 17–3 Contents of Bus Array

Field	Use
busarray$q_hw_id	Hardware ID.
busarray$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.

17.3 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 17–4.

Table 17–4 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 17–4 (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.

17.4 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 17–5.

Table 17–5 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 parameter 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.
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.

(continued on next page)

Table 17–5 (Cont.) Contents of Controller Register Access Mailbox

Field	Use
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 17–5 (Cont.) Contents of Controller Register Access Mailbox

Field	Use	
cram$l_cram_flags	The following bits are defined within cram$l_cram_flag	
	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.	

(continued on next page)

Table 17–5 (Cont.) Contents of Controller Register Access Mailbox

Field	Use
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$b_hose_num`. Otherwise, it places a zero in this field.
cram$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.

(continued on next page)

Table 17–5 (Cont.) Contents of Controller Register Access Mailbox

Field	Use	
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.	

17.5 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 17–6.

Table 17–6 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.

(continued on next page)

Table 17–6 (Cont.) Contents of Channel Request Block

Field	Use
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.
crb$b_mask	Mask that describes controller status. The following fields are defined in `crb$b_mask`: crb$v_bsy — Busy bit. IOC$PRIMITIVE_REQCHANy 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.

(continued on next page)

Table 17–6 (Cont.) Contents of Channel Request Block

Field	Use
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.
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.

(continued on next page)

Table 17–6 (Cont.) Contents of Channel Request Block

Field	Use
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 17.6) 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.

17.6 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.

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 17–7 describes the contents of the VEC substructure.

Table 17–7 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.

17.7 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 17–8.

Table 17–8 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.
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

(continued on next page)

Table 17–8 (Cont.) Contents of Device Data Block

Field	Use
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.

17.8 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 17–9 describes the fields in the driver dispatch table.

Table 17–9 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.

(continued on next page)

Table 17–9 (Cont.) Contents of Driver Dispatch Table

Field	Use
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.
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.

(continued on next page)

Table 17–9 (Cont.) Contents of Driver Dispatch Table

Field	Use
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$DEVICERR, and ERL$DEVICT 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$l_mntv_sssc	Procedure value of the routine that is called when mount verification is performed for a shadow-set state change. The DDTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the **mntv_sssc** argument to the macro. Use of this field is reserved to Digital.
ddt$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.

(continued on next page)

Table 17–9 (Cont.) Contents of Driver Dispatch Table

Field	Use
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.

(continued on next page)

Table 17–9 (Cont.) Contents of Driver Dispatch Table

Field	Use
ddt$is_stack_bcnt	Size in bytes of the kernel process stack, as indicated by the kp_stack_size macro in the DRIVER$INIT_TABLES routine. EXE$KP_STARTIO uses this value, or kpb$k_min_io_stack (currently 8 KB), whichever is larger, to determine the size of the stack created for the driver's start I/O kernel process thread.
ddt$is_reg_mask	Kernel process register save mask, as indicated by the ini_ddt_kp_reg_mask macro in the DRIVER$INIT_TABLES routine. 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 the kpb$is_reg_mask field for the lifetime of the kernel process. Note that R0, R1, R16 through R25, R28, R30, and R31 are never preserved and are illegal in a register save mask. OpenVMS represents the set of these registers by the symbolic constant KPREG$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 ini_ddt_kp_startio macro in the DRIVER$INIT_TABLES routine. EXE$KP_STARTIO calls the start-I/O routine specified in this field after setting up the kernel process environment.

17.9 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 17–10 describes the driver prologue table.

Table 17–10 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.
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.

(continued on next page)

Table 17–10 (Cont.) Contents of Driver Prologue Table

Field	Use
dpt$w_maxunits	Maximum number of units on controller that this driver supports. Specify this value in the ini_dpt_maxunits macro in the DRIVER$INIT_TABLES routine. If no value is specified, the default is eight units.
dpt$w_ucbsize	Size in bytes of the unit control block for a device that uses this driver. Every driver must specify a value for this field in the ini_dpt_ucbsize macro in the DRIVER$INIT_TABLES routine. OpenVMS supplies the symbolic constants described in Table 17–17 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$iw_idb_crams	Number of CRAMS to be allocated and associated with the IDB. The driver-loading procedure allocates the number of CRAMs specified in the ini_dpt_idb_crams macro in the DRIVER$INIT_TABLES routine 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 the ini_dpt_ucb_crams macro in the DRIVER$INIT_TABLES routine and inserts them in the linked list headed by ucb$ps_cram.

(continued on next page)

Table 17–10 (Cont.) Contents of Driver Prologue Table

Field	Use	
dpt$l_flags	Driver-loading flags. The driver can specify any of a set of flags as the value of the **flags** argument to the DPTAB macro. The driver-loading procedure modifies its loading and reloading algorithm based on the settings of these flags.	
	The following bits are defined within the `dpt$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.

(continued on next page)

Table 17–10 (Cont.) Contents of Driver Prologue Table

Field	Use	
	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 `ini_dpt_maxunits` macro in the DRIVER$INIT_TABLES routine which is the maximum units specified in the SYSMAN command IO CONNECT.
	dpt$v_scsi_port	Driver is a SCSI port driver.
dpt$il_adptype	Type of adapter used by the devices using this driver. The DRIVER$INIT_TABLES routine uses the contents of the `ini_dpt_adapt` macro 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.	
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.	

(continued on next page)

Table 17–10 (Cont.) Contents of Driver Prologue Table

Field	Use
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$w_defunits. The `ini_dpt_defunits` macro in the DRIVER$INIT_TABLES routine inserts a procedure value in this field when the driver specifies the routine's address in the **deliver** argument to the macro.
dpt$ps_unload	Procedure value of the driver routine to be called when driver is reloaded. The DPTAB macro inserts a procedure value in this field when the driver specifies the routine's address in the `unload` parameter to the macro. The driver-loading procedure calls the driver unloading routine before reinitializing all device units associated with the driver. Note that driver reloading is not yet supported on OpenVMS Alpha systems.
dpt$ps_ddt	Address of DDT.
dpt$ps_ddb_list	Header of singly-linked list of DDBs serviced by this driver. This field contains the address of the first DDB in the list. The `DDB$PS_DRVLINK` field 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.

(continued on next page)

Table 17–10 (Cont.) Contents of Driver Prologue Table

Field	Use
dpt$l_vector	Address of a driver-specific vector table. A terminal class or port driver stores the address of its class or port entry vector table in this field. For example, a terminal port driver uses this cell as a pointer to a table of addresses within the driver containing the procedure values of routines in the port driver that are called by the terminal class driver.
dpt$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`.

(continued on next page)

Table 17–10 (Cont.) Contents of Driver Prologue Table

Field	Use
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.

17.10 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 17–11 describes the interrupt dispatch block.

Table 17–11 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$w_units	Maximum number of units connected to the controller. The maximum number of units is specified in the `ini_dpt_defunits` macro in the DRIVER$INIT_TABLES routine 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 17–11 (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)

(continued on next page)

Table 17–11 (Cont.) Contents of Interrupt Dispatch Block

Field	Use
idb$l_device_specific	Longword field available to drivers for device-specific purposes.
idb$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.

17.11 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 17–12. 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 17–12 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.

(continued on next page)

Table 17–12 (Cont.) Contents of I/O Request Packet (IRP)

Field	Use
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$L_SVAPTE` to unlock the buffer pages locked for the entire transfer.
irp$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.

(continued on next page)

Table 17–12 (Cont.) Contents of I/O Request Packet (IRP)

Field	Use
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 17–12 (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`)

(continued on next page)

Table 17–12 (Cont.) Contents of I/O Request Packet (IRP)

Field	Use
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$v_cacheio — I/O using VBN cache buffers.
irp$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.

(continued on next page)

Table 17–12 (Cont.) Contents of I/O Request Packet (IRP)

Field	Use
irp$l_boff	Byte offset into the first (or current) page of a direct-I/O transfer. FDT routines calculate this offset and write its value into this field and into irp$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$FINISHIO(write the contents of R0 into this field. The I/O postprocessing routine copies the contents of this field into the user's IOSB. EXE$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 $QIO request parameter. Note that, when error logging is enabled, the contents of irp$l_media is copied into an EMB as the "disk size".
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.

(continued on next page)

Table 17–12 (Cont.) Contents of I/O Request Packet (IRP)

Field	Use
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.

(continued on next page)

Table 17–12 (Cont.) Contents of I/O Request Packet (IRP)

Field	Use	
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$l_keydesc	Address of encryption key.	
irp$l_qio_p*n*	Function-specific $QIO system service arguments (**p1** through **p6**). EXE$QIO copies these arguments to the appropriate IRP fields.	

17.12 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 17–13 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$MODIFYLOCK_ERR). If an error occurs during the locking procedure, the driver must unlock all previously locked regions using MMG_STD$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 17–13.

Table 17–13 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.
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.

17.13 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 the `ini_ddt_kp_startio` in the DRIVER$INIT_TABLES routine, or by explicitly invoking KP_ALLOCATE_KPB or calling EXE$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 17–14. Table 17–15 describes the debugging area.

Table 17–14 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.

(continued on next page)

Table 17–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use	
kpb$ib_subtype	Type of data structure. EXE$KP_ALLOCATE_KPB writes the symbolic constant DYN$C_KPB in this field when it creates the KPB.	
kpb$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$K_MIN_IO_STACK (currently 8 KB), rounded up to a multiple of CPU-specific pages.	
kpb$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.

(continued on next page)

Table 17–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use	
	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$ps_saved_sp	Previous stack pointer. When a kernel process has been started or resumed, this field contains the value of the SP register when the executing thread is preempted (but after the values indicated by the `ini_ddt_kp__reg_mask` macro have been pushed onto the stack). EXE$KP_STALL_GENERAL restores this value to the SP register when the kernel process is suspended.	
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`).	
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.	

(continued on next page)

Table 17–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use
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$ps_spl_stall_rtn. You must use the address in this field to locate the spin lock area for FGT KPBs, using the prefix code_example>(kpbspl$) in place of code_example>(kpb$) in the symbolic offsets to the spin lock area's fields.
kpb$ps_dbg_ptr	Address of the KPB debugging area. EXE$KP_ALLOCATE_KPB writes this field when creating the KPB. See Table 17–15 for a a description of the KPB debugging area. VEST KPBs do not typically include the debugging area.
kpb$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 17–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use
kpb$is_prm_length	Length of the KPB parameter area, as indicated in the `param_length` parameter to EXE$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).

(continued on next page)

Table 17–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use
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$ps_fqbl, fork-queue backward link.
kpb$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.

(continued on next page)

Table 17–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use
kpb$ib_rqtype	Type of TQE. Before using this section of the KPB as an embedded TQE, you must indicate the TQE type in this field. Alternatively, as kpb$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:SYS$LIB_C.TLB) for the appropriate spin lock index.
kpb$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.

(continued on next page)

Table 17–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use
kpb$is_timeout_time	Timeout for wait-for-interrupt operation. IOC$KP_WFIKPCH and IOC$KP_WFIRLCH initialize this field, which is used by the corresponding scheduling stall routine when calling the appropriate basic OpenVMS suspension routine. Note that this field exists only in VEST KPBs.
kpb$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. OpenVMS 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).

(continued on next page)

Table 17–14 (Cont.) Contents of Kernel Process Block (KPB)

Field	Use	
	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$is_newipl	Location in which the SCSI port drivers save the current IPL when invoking the DEVICELOCK macro to synchronize access to a device's database, and from which they restore IPL when invoking the DEVICEUNLOCK macro. Note that this field exists only in VEST KPBs.	
kpb$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 17–15 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.

17.14 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 17–16.

Table 17–16 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.
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.

(continued on next page)

Table 17–16 (Cont.) Contents of Object Rights Block

Field	Use	
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.	

17.15 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 17–18. 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 17–17 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 17–17 UCB Extensions and Sizes Defined in $UCBDEF

Extension	Used by	Size	Table
Base UCB	All devices	ucb$k_size	17–18
Error log extension	All disk and tape devices	ucb$k_erl_length	17–19
Dual-path extension	Reserved to Digital	ucb$k_2p_length	—
Local tape extension \ 17–20)	All tape devices	ucb$k_lcl_tape_length	value
Local disk extension	All disk devices	ucb$k_lcl_disk_length	value

(continued on next page)

Table 17–17 (Cont.) UCB Extensions and Sizes Defined in $UCBDEF

Extension	Used by	Size	Table
\ 17–21)			
Terminal extension[1]	Terminal class and port drivers	ucb$k_tt_length	17–22

[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,-
    .
    .
    .
```

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.

Table 17–18 describes the contents of the unit control block.

Table 17–18 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.
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 17–18 (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$l_lockid	Lock management lock ID of device allocation lock. A lock management lock is used for device allocation so that device allocation functions properly for cluster-accessible devices in an Open VMScluster (dev$v_clu set within ucb$l_devchar2).

(continued on next page)

Table 17–18 (Cont.) Contents of Unit Control Block

Field	Use
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.
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.

(continued on next page)

Table 17–18 (Cont.) Contents of Unit Control Block

Field	Use
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:SYS$LIB_C.TLB) 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$V_GEN — Generic device
	DEV$V_AVL — Device available for use
	DEV$V_MNT — Device mounted
	DEV$V_MBX — Mailbox device
	DEV$V_DMT — Device marked for dismount

(continued on next page)

Table 17–18 (Cont.) Contents of Unit Control Block

Field	Use	
	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:SYS$LIB_C.TLB) for this field. The driver-loading procedure writes the field when the procedure creates the UCB.	
	The system defines the following device characteristics:	
	DEV$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$V_NNM	Device name has a prefix of the format "*node*$"

(continued on next page)

Table 17–18 (Cont.) Contents of Unit Control Block

Field	Use	
	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
	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.	

(continued on next page)

Table 17–18 (Cont.) Contents of Unit Control Block

Field	Use
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:SYS$LIB_C.TLB) for this field. The driver-loading procedure writes this field when it creates the UCB. Drivers with set mode and device characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request. VMS defines the following device classes: DC$_DISK — Disk DC$_TAPE — Tape DC$_SCOM — Synchronous communications DC$_CARD — Card reader DC$_TERM — Terminal DC$_LP — Line printer DC$_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

(continued on next page)

Table 17–18 (Cont.) Contents of Unit Control Block

Field	Use
ucb$b_devtype	Device type. The DPT of every driver should specify a symbolic constant (defined by the $DCDEF macro in SYS$LIBRARY:SYS$LIB_C.TLB) for this field. The driver-loading procedure writes the field when it creates the UCB. Drivers for devices with set mode and set characteristics functions can rewrite the value in this field with data supplied in the characteristics buffer, the address of which is passed in the I/O request.
ucb$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.
ucb$l_ioqbl	Pending-I/O queue listhead backward link. EXE$INSERTIRP and IOC$REQCOM modify the pending-I/O queue.

(continued on next page)

Table 17–18 (Cont.) Contents of Unit Control Block

Field	Use
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.

(continued on next page)

Table 17–18 (Cont.) Contents of Unit Control Block

Field	Use
ucb$l_amb	Associated mailbox UCB pointer. A spooled device uses this field for the address of its associated device. Devices that are nonshareable and not file oriented can use this field for the address of an associated mailbox.
ucb$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.

(continued on next page)

Table 17–18 (Cont.) Contents of Unit Control Block

Field	Use	
	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 17–18 (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 the `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.

(continued on next page)

Table 17–18 (Cont.) Contents of Unit Control Block

Field	Use	
	ucb$v_dx_write	Console floppy write operation.
	ucb$v_datacache	Data blocks are being cached.
	Terminal class and port drivers use bits in the ucb$l_devsts field as follows:	
	ucb$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$l_ioqfl).	
ucb$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.	

(continued on next page)

Table 17–18 (Cont.) Contents of Unit Control Block

Field	Use
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 the `ucb$l_svpn` field when the procedure creates the UCB. Disk drivers use this field for ECC error correction.
ucb$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 17–18 (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 17–19 describes the contents of the UCB error log extension.

Table 17–19 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 17–20 describes the contents of th UCB local tape extension.

Table 17–20 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 17–21 describes the contents of the UCB local disk extension.

Table 17–21 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 17–22 describes the contents of the UCB terminal extension.

Table 17–22 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.

(continued on next page)

Table 17–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
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
	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

(continued on next page)

Table 17–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
	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

(continued on next page)

Table 17–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
	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.	
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

(continued on next page)

Table 17–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
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.
	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.	

(continued on next page)

Table 17–22 (Cont.) Contents of UCB Terminal Extension

Field	Use
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 17–22 (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.

(continued on next page)

Table 17–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
	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$v_pc_nocrlf	No auto line feed. If set, a line feed is not generated following a carriage return.
	tty$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.

(continued on next page)

Table 17–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
	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$m_unloop	Reset loopback mode.
	io$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$m_line_on	Reenable interrupts on this line.
	`ucb$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.	

(continued on next page)

Table 17–22 (Cont.) Contents of UCB Terminal Extension

Field	Use	
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.

17.16 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 `idb$l_vector` field contains the vector offset (into the SCB or the ADP vector table).

- For devices with multiple interrupt vectors, the idb$l_vector field 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 17–23.

Table 17–23 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.

18

Device Driver Entry Points

This chapter describes the standard driver routines used as entry points in an OpenVMS Alpha device driver.

Alternate Start-I/O Routine

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

Prototype

void driver_altstart_routine (IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Essentials

Identifying the Routine

Specify the address of the alternate start-I/O routine by using the ini_ddt_altstart macro in the driver$init_tables ( ) routine. For example:

```
ini_ddt_altstart (&driver$fdt, driver_altstart).
```

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.

Cancel-I/O Routine

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

Prototype

void driver_cancel (CHAN *chan, IRP *irp, PCB *pcb, UCB * ucb, int reason)

Parameters

Name	Access	Description
chan	Input	Pointer to the channel index number.
irp	Input	Pointer to the I/O request packet, if any, for device (contents of UCB$L_IRP).
pcb	Input	Pointer to the process control block of process for which the I/O request is being canceled.
ucb	Input	Pointer to the unit control block.
reason	Input	Reason for cancellation can be one of the following: CAN$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 `ini_ddt_cancel` macro in the DRIVER$INIT_TABLES routine. The macro places the procedure value of this routine into the DDT. Many drivers specify the system routine IOC_STD$CANCELIO as their cancel-I/O routine.

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.

Cancel Selective Routine

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

Prototype

```
int driver_cancel_selective (PCB *pcb, UCB *ucb, int chan, int iosb_vector, int
    iosb_count)
```

Parameters

Name	Access	Description
pcb	Input	Pointer to the process control block of process for which the I/O request is being canceled.
ucb	Input	Pointer to the unit control block.
chan	Input	Pointer to the channel index number.
iosb_vector	Input	Pointer to the vector of address of I/O status blocks (IOSBs), or zero.
iosb_count	Input	Pointer to the number of addresses in the IOSB vector.

Essentials

Identifying the Routine

Supply the address of the cancel selective routine in the `ini_ddt_cancel_selecti` macro in the DRIVER$INIT_TABLES routine. The macro places the procedure value of this routine into the DDT.

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.

Channel Assign Routine

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

Prototype

void driver_channel (UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
ucb	Input	Pointer to the unit control block.
ccb	Input	Pointer to the channel control block.

Essentials

Identifying the Routine

Supply the address of the channel assign routine in the `ini_ddt_channel_assign` macro in the DRIVER$INIT_TABLES routine. The macro places the procedure value into the DDT.

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.

Cloned UCB Routine

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

Prototype

```
int driver_cloneducb (UCB *cloned_ucb, DDT *ddt, PCB *pcb, UCB *template_ucb)
```

Parameters

Name	Access	Description
cloned_ucb	Input	Pointer to the cloned unit control block. 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	Input	Pointer to the driver dispatch table.
pcb	Input	Pointer to the process control block of the current process.
template_ucb	Input	Pointer to the template unit control block.

Parameter Fields

Field	Contents
cloned_ucb->	
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
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

Essentials

Identifying the Routine

Specify the address of a cloned UCB routine in the `ini_ddt_cloneducb` macro in the DRIVER$INIT_TABLES routine. Only drivers for template devices, such as mailboxes, specify a cloned UCB routine.

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.

Controller Initialization Routine

Prepares a controller for operation.

Prototype

int driver_ctrlinit (IDB *idb, DDB *ddb, CRB *crb)

Parameters

Name	Access	Description
idb	Input	Pointer to the interrupt dispatch block associated with the controller.
ddb	Input	Pointer to the device data block associated with the controller.
crb	Input	Pointer to the controller request block.

Essentials

Identifying the Routine

Specify the address of a controller initialization routine in the `ini_ddt_ctrlinit` parameter in the DRIVER$INIT_TABLES routine.

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.

CSR Mapping Routine

Maps the device registers, which reside in I/O space, into system virtual address space.

Prototype

int csr_mapping_routine (IDB *idb, DDB *ddb, CRB *crb)

Parameters

Name	Access	Description
idb	Input	Pointer to the IDB for the device.
ddb	Input	Pointer to the DDB for the device.
crb	Input	Pointer to the CRB for the device.

Essentials

Return value

The routine must return a successful status for the device initialization to continue. If a unsuccessful status value is returned by the driver's CSR mapping routine, the controller and unit initialization routines will not be called.

Specified by

`ini_ddt_csr_mapping` macro in the DRIVER$INIT_TABLES routine.

Description

The CSR mapping routine is called at IPL 8 with the IOLOCK8 spinlock held. This environment allows the driver CSR mapping routine to call the IOC$MAP_IO routine to map the device registers. In contrast, IOC$MAP_IO cannot be called from the device driver's controller or unit initialization routines because they are called at IPL 31.

The CSR mapping routine is called during the device driver initialization sequence after the devices I/O database structures are completed and linked into the I/O database but before the device driver controller initialization routine.

Driver Channel Grant Fork Routine

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.

Prototype

void driver_chn_grant (IRP *irp, IDB *idb, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
idb	Input	Pointer to the interrupt dispatch block (IDB).
ucb	Input	Pointer the unit control block of the device assigned to the process I/O channel.

Driver Device Timeout Routine

The driver device timeout routine is the wait-for-interrupt timeout routine. The EXE$TIMEOUT routine calls this routine when an operation set up by the `wfikpch` or `wfirlch` macros take more than the specified number of seconds.

This routine is called at device IPL with both the fork spinlock and the device spinlocks held.

Prototype

```
void driver_timeout (IRP *irp, int64 fr4, UCB *ucb)
```

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet from ucb->ucb$q_fr3.
fr4	Input	The 64-bit value from ucb->fkb$q_fr4.
fr4	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Device Data Structure Initialization Routine

Called once for each unit by the $LOAD_DRIVER service after that UCB is created. At the point of this call the UCB has not yet been fully linked into the I/O database. This routine is responsible for filling in driver specific fields that in the I/O database structures that are passed as parameters to this routine.

Prototype

void struc_init (CRB *crb, DDB *ddb, IDB *idb, ORB *orb, UCB *ucb)

Parameters

Name	Access	Description
crb	Input	Pointer to the associated controller request block.
ddb	Input	Pointer to the associated device data block.
idb	Input	Pointer to the associated interrupt dispatch block.
orb	Input	Pointer to the associated object rights block.
ucb	Input	Pointer to the unit control block that is to be initialized.

Description

After the device data structure initialization routine routine is called for a new unit, the reinitialization routine is also called. The $LOAD_DRIVER service then completes the integration of these device specific structures into the I/O database.

Note that this routine must confine its actions to filling in these I/O database structures and may not attempt to initialize the hardware device. Initialization of the hardware device is the responsibility of the controller and unit initialization routines that are called some time later.

Device I/O Data Structure Re-initialization Routine

Called once for each unit by the $LOAD_DRIVER service immediately after the structure initialization routine is called. Because this routine is called once for each unit by the $LOAD_DRIVER service when a driver image is reloaded, it fills in the fields in the I/O database structures that point to this driver image. Note that this routine must confine its actions to filling in these I/O database structures.

Prototype

```
void struc_reinit (CRB *crb, DDB *ddb, IDB *idb, ORB *orb, LR_UCB *ucb)
```

Parameters

Name	Access	Description
crb	Input	Pointer to associated controller request block.
ddb	Input	Pointer to associated device data block.
idb	Input	Pointer to associated interrupt dispatch block.
orb	Input	Pointer to associated object rights block.
ucb	Input	Pointer to the unit control block that is to be initialized.

Driver Resume from Interrupt Routine Entry

Continues the processing of an I/O request in the context of the device driver's interrupt service routine. This routine is specified by either the `wfikpch` or `wfirlch` macros. This routine is invoked by using the `rfi` macro in the device driver interrupt service routine.

The driver resume from interrupt routine executes at device IPL with the device spinlock held.

Note that 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. The driver fork routine would then be resumed from the interrupt service routine by:

```
ucb->ucb$v_tim = 0;
exe_std$queue_fork ( (FKB *) ucb );
```

Prototype

void driver_resume_fi (IRP *irp, int64 fr4, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet from ucb->ucb$q_fr3
fr4	Input	Pointer to the 64-bit value from ucb->fkb$q_fr4.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Note

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.

Driver Table Initialization Routine

Completes the initialization of the DPT, DDT, and FDT structures. If a driver image contains a routine named DRIVER$INIT_TABLES, this routine is called once by the $LOAD_DRIVER service immediately after the driver image is loaded or reloaded and before any validity checks are performed on the DPT, DDT, and FDT. A prototype version of these structures is built into this image at link time from the VMS$VOLATILE_PRIVATE_INTERFACES.OLB library.

Prototype

```
int driver$init_tables ( )
```

Implicit Inputs

Name	Access	Description
driver$dpt	Input	Externally defined name for the prototype DPT structure that is linked into the driver.
driver$ddt	Input	Externally defined name for the prototype DDT structure that is linked into the driver.
driver$fdt	Input	Externally defined name for the prototype DDT structure that is linked into the driver.

Description

Note that the device related data structures (that is, DDB, UCB, etc.) have not yet been created when the Initialize Driver Tables routine is called. routine is called. Therefore, the actions of this routine must be confined to the initialization of the DPT, DDT, and FDT structures that are contained in the driver image.

FDT Upper-Level Action Routine

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

Prototype

int driver_fdt_routine (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet for the current I/O request.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process-I/O channel specified as a parameter to the $QIO request.
ccb	Input	Pointer to the channel control block that describes the process-I/O channel.

Parameter Fields

Field	Contents
irp->	
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.

Essentials

Identifying the Routine

Use the `ini_fdt_act` macro in the DRIVER$INIT_TABLES routine 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. For example:

```
ini_fdt_act (&driver$fdt, IO$_SETMODE, driver_fdt_routine, buffered)
```

This macro also specifies if the I/O function is buffered or direct.

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 parameter 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.

FDT Error-Handling Callback Routine

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

Prototype

int driver_errtn_routine (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb, int status)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet for the current I/O request.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process-I/O channel specified as a parameter to the $QIO request.
ccb	Input	Pointer to the channel control block that describes the process-I/O channel.
status	Input	Pointer to the error status returned by buffer accessibility check (SS$_ACCVIO or SS$_BADPARAM) or buffer locking operation (SS$_ACCVIO, SS$_INSFWSL, or page fault status).

Essentials

Identifying the Routine

Use the `errtn` parameter in a call to EXE_STD$MODIFYLOCK, EXE_STD$READLOCK, or EXE_STD$WRITELOCK.

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.

Interrupt Service Routine

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

Prototype

void driver_isr (IDB *idb, int scb_offset)

Parameters

Name	Access	Description
idb	Input	Pointer to the interrupt dispatch block.
scb_offset	Input	Pointer to the ???

Essentials

Identifying the Routine

Devices require an interrupt service routine for each interrupt vector. Use the dpt_store_isr macro in the structure reinitialization routine. For example, use `dpt_store_isr (crb, driver_isr)` 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.

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 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);
```

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);
```

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.

Mount Verification Routine

Performs device-specific mount verification.

Prototype

int driver_mntver_routine (IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet, or zero to complete mount verification.
ucb	Input	Pointer to the unit control block.

Essentials

Identifying the Routine

Supply the address of the mount verification routine in the `ini_ddt_mntv_for` macro in the DRIVER$INIT_TABLES routine. The default value of this macro is the only value allowed for device drivers not supplied by Digital.

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.

Register Dumping Routine

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

Prototype

void driver_regdump_routine (int buffer, arg_2, UCB *ucb)

Parameters

Name	Access	Description
buffer	Input	Pointer to the address of buffer into which a register dumping routine copies the contents of device registers.
arg_2	Input	Pointer to the device-specific argument, usually a controller register access mailbox (CRAM).
ucb	Input	Pointer to the Unit control block.

Essentials

Identifying the Routine

Specify the name of the register dumping routine in the `ini_ddt_regdmp` macro in the DRIVER$INIT_TABLES routine.

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

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

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

Prototype

void driver_startio (IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block.

Parameter Fields

Field	Contents
ucb->	
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 `ini_ddt_start` macro in the DRIVER$INIT_TABLES routine.

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.

Start-I/O Routine (Kernel Process)

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

Prototype

void driver_kpstartio (KPB *kpb)

Parameters

Name	Access	Description
kpb	Input	Pointer to the kernel process block.

Essentials

Identifying the Routine

Specify the name of the kernel process start-I/O routine (EXE_STD$KP_STARTIO) in the `ini_ddt_kp_startio` macro in the DRIVER$INIT_TABLES routine.

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.

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.

Prototype

void driver_timeout_routine (IRP *irp, int64 fr4, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the pointer to the IRP from ucb->ucb$q_fr3.
fr4	Input	Pointer to the 64-bit value from ucb->fkb$q_fr4.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Essentials

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.

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.

Prototype

```
void driver_deliver (DDB *ddb, IDB idb, int unit_number, int scratch_area, ADP
    *adp)
```

Parameters

Name	Access	Description
ddb	Input	Pointer to the device data block.
idb	Input	Pointer to the interrupt dispatch block; 0 if none exists.
unit_number	Input	Pointer to the number of unit that the unit delivery routine must decide to configure or not to configure.
scratch_area	Input	Pointer to the address of quadword scratch area.
adp	Input	Pointer to the adapter control block.

Essentials

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.

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.

Unit Initialization Routine

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

Prototype

int driver_unit_init (IDB *idb, UCB *ucb)

Parameters

Name	Access	Description
idb	Input	Pointer to the interrupt dispatch block associated with the controller.
ucb	Input	Pointer to the unit control block.

Essentials

Identifying the Routine

Specify the address of the unit initialization routine `ini_ddt_unitinit` macro in the DRIVER$INIT_TABLES routine.

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.

19

System Routines

This chapter describes many of the operating system routines commonly used by OpenVMS Alpha device drivers.

The function prototypes for the routines used by OpenVMS Alpha device drivers are available in the following files in SYS$LIBRARY:SYS$LIB_C.TLB:

```
#include <acp_routines.h>
#include <com_routines.h>
#include <erl_routines.h>
#include <exe_routines.h>
#include <ioc_routines.h>
#include <ldr_routines.h>
#include <lnm_routines.h>
#include <mmg_routines.h>
#include <sch_routines.h>
#include <smp_routines.h>
```

ACP_STD$ACCESS

FDT routine that handles the access and create I/O function codes for device drivers that use an ACP or the XQP.

Prototype

int acp_std$access (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

ACP_STD$ACCESSNET

Connects to network function processing.

Prototype

int acp_std$accessnet (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

ACP_STD$DEACCESS

Deaccesses ACP function processing.

Prototype

int acp_std$deaccess (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

ACP_STD$MODIFY

Deletes and modifies ACP function processing.

Prototype

int acp_std$modify (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

ACP_STD$MOUNT

Initiates ACP mount function processing.

Prototype

int acp_std$mount (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

ACP_STD$READBLK

Processes a read block ACP function.

Prototype

int acp_std$readblk (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

ACP_STD$WRITEBLK

Processes a write block ACP function.

Prototype

int acp_std$writeblk (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

COM_STD$DELATTNAST

Delivers all attention ASTs linked in the specified list.

Prototype

```
void com_std$delattnast (ACB **acb_lh, UCB *ucb)
```

Parameters

Name	Access	Description
acb_lh	Input	Pointer to the listhead of AST control blocks.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

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.

COM_STD$DELATTNASTP

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

Prototype

```
void com_std$delattnastp (ACB **acb_lh, UCB *ucb, int ipid)
```

Parameters

Name	Access	Description
acb_lh	Input	Pointer to the listhead of AST control blocks.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ipid	Input	Pointer to the 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.

COM_STD$DELCTRLAST

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

Prototype

```
void com_std$delctrlast (ACB **acb_lh, UCB *ucb, int matchchar, int32
    *inclchar_p)
```

Parameters

Name	Access	Description
acb_lh	Input	Pointer to the listhead of AST control blocks.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
matchchar	Input	Pointer to a match character.
inclchar_p	Output	Pointer to the 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.

COM_STD$DELCTRLASTP

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

Prototype

```
void com_std$delctrlastp (ACB **acb_lh, UCB *ucb, int ipid, int matchchar, int32
    *inclchar_p)
```

Parameters

Name	Access	Description
acb_lh	Input	Pointer to the listhead of AST control blocks.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ipid	Input	Pointer to the internal process ID (IPID) for the target process.
matchchar	Input	Pointer to a match character.
inclchar_p	Output	Pointer to the 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.

COM_STD$DRVDEALMEM

Deallocates system dynamic memory.

Prototype

void com_std$drvdealmem (void *ptr)

Parameters

Name	Access	Description
ptr	Input	Pointer to the block to be allocated. 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. The FKB$ symbols are defined by the fkbdef.h header file in SYS$LIB_C.TLB.

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.

COM_STD$FLUSHATTNS

Removes specified ASTs from an attention AST list.

Prototype

int com_std$flushattns (PCB *pcb, UCB *ucb, int chan, ACB **acb_lh)

Parameters

Name	Access	Description
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
chan	Input	Pointer to the number of the assigned I/O channel.
acb_lh	Input	Pointer to the listhead of AST control blocks.

Parameter Fields

Field	Contents
pcb->	
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.

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.

COM_STD$FLUSHCTRLS

Removes specified ASTs from a control AST list.

Prototype

int com_std$flushctrls (PCB *pcb, UCB *ucb, int chan, ACB **acb_lh, int32 *mask_p)

Parameters

Name	Access	Description
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
chan	Input	Pointer to the number of the assigned I/O channel.
acb_lh	Input	Pointer to the listhead of AST control blocks.
mask_p	Input	Pointer to the summary mask of active control characters. COM_STD$FLUSHCTRLS updates this mask.

Parameter Fields

Field	Contents
pcb->	
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.

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.

COM_STD$POST, COM_STD$POST_NOCNT

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

Prototype

void com_std$post (IRP *irp, UCB *ucb)

void com_std$post_nocnt (IRP *irp)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
ucb	Input	(COM_STD$POST only) Pointer to the unit control block of the device assigned to the process I/O channel.

Parameter Fields

Field	Contents
irp->	
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

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.

COM_STD$SETATTNAST

Enables or disables attention ASTs.

Prototype

int com_std$setattnast (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb, ACB **acb_lh)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for the process I/O channel.
acb_lh	Input	Pointer to the listhead of AST control blocks.

Parameter Fields

Field	Contents
irp->	
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->	
PCB$L_ASTCNT	Number of ASTs remaining in process quota. COM_STD$SETATTNAST decreases PCB$L_ASTCNT if it successfully queues the AST.
PCB$L_PID	Process ID

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.

COM_STD$SETCTRLAST

Enables or disables control ASTs.

Prototype

```
int com_std$setctrlast (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb, ACB **acb_lh,
    int mask, TAST **tast_p)
```

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
acb_lh	Input	Pointer to the listhead of AST control blocks.
mask_p	Input	Pointer to the summary mask of active control characters.
tast_p	Output	Pointer to the address of the control AST block (TAST), returned as output from COM_STD$SETCTR.

Parameter Fields

Field	Contents
irp->	
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->	
PCB$L_ASTCNT	Number of ASTs remaining in process quota. COM_STD$SETCTRLAST decreases PCB$L_ASTCNT if it successfully queues the AST.
PCB$L_PID	Process ID.

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.

ERL_STD$ALLOCEMB

Allocates an error log message buffer and initializes its header.

Prototype

```
int erl_std$allocemb (int size, EMBDV **embdv_p)
```

Parameters

Name	Access	Description
size	Input	Pointer to the size of the error message buffer in bytes.
embdv_p	Output	Address of a pointer in which ERL_STD$ALLOCEM writes the address of the error message buffer.

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.

ERL_STD$DEVICEATTN, $DEVICERR,DEVICTMO

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

Prototype

void erl_std$deviceattn (int64 driver_param, UCB *ucb)

void erl_std$devicerr (int64 driver_param, UCB *ucb)

void erl_std$devictmo (int64 driver_param, UCB *ucb)

Parameters

Name	Access	Description
driver_param	Input	Pointer to the Parameter to be passed to the register dumping routine, usually a controller register access mailbox (CRAM).
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Parameter Fields

Field	Contents
ucb-> write fields	
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$W_ERRORBUF contains the size of the error message buffer in bytes.
ucb-> read fields	
UCB$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.

ERL_STD$RELEASEMB

Releases an error message buffer to the error logging process.

Prototype

```
void erl_std$releasemb (EMBDV *embdv)
```

Parameters

Name	Access	Description
embdv	Input	Pointer to the 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.

EXE$BUS_DELAY

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

Prototype

int exe$bus_delay (ADP *adp)

Parameters

Name	Access	Description
adp	Input	Pointer to the address of the ADP.

Context

EXE$BUS_DELAY conforms to the OpenVMS Calling Standard.

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.

Prototype

int exe$delay (int64 *delay_nanos)

Parameters

Name	Access	Description
delta	Input	Delay time specified in nanoseconds.

Context

EXE$DELAY conforms to the OpenVMS calling standard.

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.

Prototype

int exe$kp_allocate_kpb (KPB **kpb_p, int stksiz, int flags, int paramsiz)

Parameters

Name	Access	Description
kpb	Input	Pointer to the address of the KPB.
stack_size	Input	Requested size (in bytes) of the kernel process stack.
flags	Input	Flags indicating the type, size, and configuration of the KBP to be created.
paramsiz	Input	Size in bytes of the KPB parameter, if any.

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.

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.

  EXE$KP_ALLOCATE_KPB accepts only the following flags:

KPB$V_VEST	KPB must be a VEST KPB. (See Chapter 17 for a description of VEST KPBs.)
KPB$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.

- 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.

Prototype

int exe$kp_deallocate_kpb (KPB *kpb)

Parameters

Name	Access	Description
kpb	Input	Pointer to the address of the 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.

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.

Prototype

```
int exe$kp_end(KPB *kpb)
```

Parameters

Name	Access	Description
kpb	Input	Pointer to the address of the 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.

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.

Prototype

int exe$kp_fork (KPB *kpb, FKB *fkb)

Parameters

Name	Access	Description
kpb	Input	Pointer to the address if 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	Input	Pointer to the 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).

Context

EXE$KP_FORK conforms to the OpenVMS Alpha calling standard. It can only be called by a 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$_NORMAL	The routine completed successfully.
SS$_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.

Prototype

int exe$kp_fork_wait (KPB *kpb, FKB *fkb)

Parameters

Name	Access	Description
kpb	Input	Pointer to the address of the kernel process block.
fkb	Input	Pointer to the address of a fork block, usually in the UCB. If this argument is omitted, EXE$KP_FORK_WAIT uses the fork block within the KPB (KPB$PS_FKBLK).

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.

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.

Prototype

int exe$kp_restart (KPB *kpb, int thread_sts)

Parameters

Name	Access	Description
kpb	Input	Pointer to the address of the kernel process block.
thread_sts	Input	Status value to be returned to the kernel process that is to be resumed. This is the status returned by the call to EXE$KP_STALL_GENERAL. If the **thread_sts** argument is not present, EXE$KP_RESTART returns SS$_NORMAL status to the kernel process.

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.

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.

Prototype

int exe$kp_stall_general (KPB *kpb)

Parameters

Name	Access	Description
kpb	Input	Pointer to the address of the kernel process block.

Context

EXE$KP_STALL_GENERAL conforms to the OpenVMS Alpha Calling Standard and can only be called by a 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	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 19–1 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.
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.

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

(continued on next page)

Table 19–1 (Cont.) Kernel Process Stall Jacket Routines and Scheduling Stall Routines

Stall Jacket Routine	Scheduling Stall Routine[1]	Action of Stall Routine
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.

Format

int exe$kp_start (KPB *kpb, void (*rout)(KPB *kpb),int64 regmask)

Parameters

Name	Access	Description
kpb	Input	Pointer to the address of the kernel process block.
rout	Input	Pointer to the procedure value of the routine to be started as the top-level routine in the kernel process.
regmask	Input	Pointer to the 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.

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.

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.

Prototype

??? ???

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.

Prototype

int exe$timedwait_complete (int64 *end_value_p)

Parameters

Name	Access	Description
end_value_p	Input	End time calculated by a previous call to EXE$TIMEDWAIT_SETUP or EXE$TIMEDWAIT_SETUP_10US.

Context

EXE$TIMEDWAIT_COMPLETE conforms to the OpenVMS Alpha Calling Standard.

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.

Format

int exe$timedwait_setup (int64 *delay_nanos, int64 *end_value_p)

int exe$timedwait_setup_10us (int64 *delay_10us, int64 *end_value_p)

Parameters

Name	Access	Description
delta	Input	Delay time specified in nanoseconds.
end_value_p	Input	End time token to be supplied as input to EXE$TIMEDWAIT_COMPLETE.

Context

EXE$TIMEDWAIT_SETUP and EXE$TIMEDWAIT_SETUP_10US conform to the OpenVMS Alpha Calling Standard.

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.

Prototype

int exe_std$abortio (IRP *irp, PCB *pcb, UCB *ucb, int qio_sts)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
qio_sts	Input	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.

EXE_STD$ALLOCBUF, EXE_STD$ALLOCIRP

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

Prototype

```
int exe_std$allocbuf (int reqsize, int32 *alosize_p, void **bufptr_p)
```

```
int exe_std$allocirp (IRP **irp_p)
```

Parameters

Name	Access	Description
reqsize	Input	Size of requested buffer in bytes (EXE_STD$ALLOCBUF only). This value should include the 12 bytes required to store header information.
alosize_p	Input	Location in which EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP write the size of the requested buffer in bytes.
bufptr_p	Input	Location in which EXE_STD$ALLOCBUF and EXE_STD$ALLOCIRP write the address of allocated buffer.
IRP	Input	Pointer to the I/O request packet.

Parameter Fields

Field	Contents
bufptr_p->	
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.

EXE_STD$ALTQUEPKT

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

Prototype

void exe_std$altquepkt (IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Parameter Fields

Field	Contents
ucb->	
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.

EXE_STD$CARRIAGE

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

Prototype

void exe_std$altquepkt (IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Context

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

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.

Prototype

```
int exe_std$chkcreacces (ARB *arb, ORB *orb, PCB *pcb, UCB *ucb)
int exe_std$chkdelacces (ARB *arb, ORB *orb, PCB *pcb, UCB *ucb)
int exe_std$chkexeacces (ARB *arb, ORB *orb, PCB *pcb, UCB *ucb)
int exe_std$chklogacces (ARB *arb, ORB *orb, PCB *pcb, UCB *ucb)
int exe_std$chkphyacces (ARB *arb, ORB *orb, PCB *pcb, UCB *ucb)
int exe_std$chkrdacces (ARB *arb, ORB *orb, PCB *pcb, UCB *ucb)
int exe_std$chkwrtacces (ARB *arb, ORB *orb, PCB *pcb, UCB *ucb)
```

Parameters

Name	Access	Description
arb	Input	Pointer to the agent rights block.
orb	Input	Pointer to the object rights block.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

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.

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.

Prototype

int exe_std$finishio (IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block of the device assigned to 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

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 r 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.

EXE$ILLIOFUNC

Aborts I/O preprocessing for an I/O function not supported a driver.

Prototype

int exe$illiofunc (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

Prototype

int exe_std$insert_irp (IRP **irp_lh, IRP *irp)

Parameters

Name	Access	Description
irp_lh	Input	Pointer to the I/O queue listhead for the device.
irp	Input	Pointer to the I/O request packet.

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.

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.

Prototype

void exe_std$insioq (IRP *irp, UCB *ucb)

void exe_std$insioqc (IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Parameter Fields

Field	Contents
ucb-> read fields	
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$L_QLEN	Length of pending-I/O queue.
ucb-> write fields	
UCB$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.

EXE_STD$IORSNWAIT

Places a process in a resource wait state if it has enabled resource waits.

Prototype

int (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb, int qio_sts, int rsn)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for the process I/O channel.
qio_sts	Input	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	Input	Pointer to the 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.

EXE_STD$LCLDSKVALID

Processes I/O functions that affect the online count and local valid status of a disk.

Prototype

int exe_std$lcldskvalid (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for the process I/O channel.

Parameter Fields

Field	Contents
ucb-> read fields	
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.
ucb-> write fields	
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$_AVAILABLE.
UCB$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

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.

EXE_STD$MNTVERSIO

Initiates a mount verification I/O request to a device.

Prototype

void exe_std$mntversio (void (*rout)( ), IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
rout	Input	Pointer to the Procedure value of action routine to postprocess the mount verification I/O request.
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

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.

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.

Prototype

```
int exe_std$modify (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)
```

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for the process I/O channel.

Parameter Fields

Field	Contents
irp-> read fields	
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 $QIO system service.
irp-> write fields	
IRP$B_CARCON	Carriage control byte (from IRP$L_QIO_P4).
IRP$L_FUNC	Logical read/write function code converted to physical.
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.

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.

EXE_STD$MODIFYLOCK

Validates and prepares a user buffer for a direct-I/O, DMA read/write operation.

Prototype

```
int exe_std$modifylock (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb, void *buf, int
    bufsiz, void (*err_rout)(IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb, int errsts))
```

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for the process I/O channel.
buf	Input	Pointer to the virtual address of buffer.
bufsiz	Input	Pointer to the number of bytes in transfer.
err_rout	Input	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.

Parameter Fields

Field	Contents
irp->	
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.

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.

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.

Prototype

int exe_std$mount_ver (int iost1, int iost2, IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
iost1	Input	First longword of I/O status.
iost2	Input	Second longword of I/O status.
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

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.

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.

Prototype

int exe_std$oneparm (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

EXE_STD$PRIMITIVE_FORK

Creates a simple fork process on the local processor.

Prototype

void exe_std$primitive_fork (int64 fr3, int64 fr4, FKB *fkb)

Parameters

Name	Access	Description
fr3	Input	Pointer to value to pass to the fork routine in FKB$Q_FR3.
fr4	Input	Pointer to value to pass to the fork routine in FKB$Q_FR4.
fkb	Input	Pointer to the address of a 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.

EXE_STD$PRIMITIVE_FORK_WAIT

Inserts a fork block on the fork-and-wait queue.

Prototype

```
void exe_std$primitive_fork_wait (int64 fr3, int64 fr4, FKB *fkb)
```

Parameters

Name	Access	Description
fr3	Input	Pointer to value to pass to the fork routine in FKB$Q_FR3.
fr4	Input	Pointer to value to pass to the fork routine in FKB$Q_FR4.
fkb	Input	Pointer to the address of a 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.

EXE_STD$QIOACPPKT

Delivers an IRP to the appropriate ACP or XQP.

Prototype

int exe_std$qioacppkt (IRP *irp, PCB *pcb, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to 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

EXE_STD$QIOACPPKT is called by, and returns to, a driver's FDT routine at IPL$_ASTDEL.

EXE_STD$QIODRVPKT

Delivers an IRP to a driver's start-I/O routine or pending-I/O queue.

Prototype

int exe_std$qiodrvpkt (IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Parameter Fields

Field	Contents
ucb->read fields	
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$L_QLEN	Length of pending-I/O queue
ucb->write fields	
UCB$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.

EXE_STD$QUEUE_FORK

TBS

Prototype

void exe_std$queue_fork (FKB *fkb)

Parameters

Name	Access	Description
fkb	Input	Pointer to the address of a fork block, usually in the UCB.

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.

Prototype

```
void exe_std$queue_fork (FKB *fkb)
```

Parameters

Name	Access	Description
pcb	Input	Pointer to the process control block of the current process.
acb	Pointer to the 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.

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.

Prototype

```
int exe_std$read (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)
```

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for the process I/O channel.

Parameter Fields

Field	Contents
irp-> read fields	
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 $QIO system service.
irp-> write fields	
IRP$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
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

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.

EXE_STD$READCHK

Verifies that a process has write access to the pages in the buffer specified in a $QIO request.

Prototype

int exe_std$readchk (IRP *irp, PCB *pcb, UCB *ucb, void *buf, int bufsiz)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
buf	Input	Pointer to the virtual address of buffer.
bufsiz	Input	Pointer to the number of bytes in transfer.

Parameter Fields

Field	Contents
irp->	
IRP$L_STS	IRP$V_FUNC set, indicating a read function
IRP$L_BCNT	Size of transfer in bytes

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.

EXE_STD$READLOCK

Validates and prepares a user buffer for a direct-I/O, DMA write operation.

Prototype

```
int exe_std$readlock (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb, void *buf, int
    bufsiz, void (*err_rout)(IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb, int errsts))
```

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for the process I/O channel.
buf	Input	Pointer to the virtual address of buffer.
bufsiz	Input	Pointer to the number of bytes in transfer.
err_rout	Input	Pointer to the 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.

Parameter Fields

Field	Contents
irp->	
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

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.

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.

Prototype

int exe_std$sensemode (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

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).

Prototype

```
int exe_std$setchar (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)
```

```
int exe_std$setmode (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)
```

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for the process I/O channel.

Parameter Fields

Field	Contents
irp-> read fields	
IRP$L_FUNC	I/O function code supplied in the $QIO request.
IRP$B_RMOD	Mode of the $QIO caller.
IRP$L_QIO_P1	$QIO system service **p1** argument, containing the device characteristics quadword.
irp-> write fields	
IRP$L_MEDIA	First longword of device characteristics.
IRP$L_MEDIA+4	Second longword of device characteristics.
ucb-> write fields	
UCB$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.

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.

EXE_STD$SNDEVMSG

Builds and sends a device-specific message to the mailbox of a system process, such as the job controller or OPCOM.

Prototype

```
int exe_std$sndevmsg (MB_UCB *mb_ucb, int msgtyp, UCB *ucb)
```

Parameters

Name	Access	Description
mb_ucb	Input	Pointer to the 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	Input	Pointer to the Message type. OPCOM message types have the prefix OPC$_ and are defined by the $OPCMSG macro in SYS$LIBRARY:STARLET.MLB.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

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.

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.

Prototype

```
int exe_std$write (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)
```

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for the process I/O channel.

Parameter Fields

Field	Contents
irp-> read fields	
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 $QIO system service.
irp->write fields	
IRP$B_CARCON	Carriage control byte (from IRP$L_QIO_P4)
IRP$L_FUNC	Logical write function code converted to physical
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

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.

EXE_STD$WRITECHK

Verifies that a process has read access to the pages in the buffer specified in a $QIO request.

Prototype

int exe_std$writechk (IRP *irp, PCB *pcb, UCB *ucb, void *buf, int bufsiz)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
buf	Input	Pointer to the virtual address of buffer.
bufsiz	Input	Pointer to the 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.

EXE_STD$WRITELOCK

Validates and prepares a user buffer for a direct-I/O, DMA read operation.

Prototype

int exe_std$writelock (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb, void *buf, int bufsiz, void (*err_rout)(IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb, int errsts))

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for the process I/O channel.
buf	Input	Pointer to the virtual address of buffer.
bufsiz	Input	Pointer to the number of bytes in transfer.
err_rout	Input	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.

Parameter Fields

Field	Contents
irp->	
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

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.

EXE_STD$WRTMAILBOX

Sends a message to a mailbox.

Prototype

int exe_std$wrtmailbox (MB_UCB *mb_ucb, int msgsiz, void *msg)

Parameters

Name	Access	Description
mb_ucb	Input	Pointer to the Mailbox unit control block. (SYS$AR_JOBCTLMB contains the address of the job controller's mailbox; SYS$AR_OPRMBX contains the address of OPCOM's mailbox.)
msgsiz	Input	Pointer to the message size.
msg	Input	Pointer to the address of the 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.

EXE_STD$ZEROPARM

Delivers an I/O request that requires no parameters to a driver's start-I/O routine.

Prototype

int exe_std$zeroparm (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

IOC$ALLOC_CNT_RES

Allocates the requested number of items of a counted resource.

Prototype

int ioc$alloc_cnt_res (CRAB *crab, CRCTX *crctx, [int64 cntxt1], int64 cntx)

Parameters

Name	Access	Description
crab	Input	Pointer to the address of CRAB that describes the counted resource. For adapters that supply a counted resource, such as map registers, ADP$L_CRAB often contains this address.
crctx	Input	Pointer to the address of CRCTX structure that describes the request for the counted resource.
cntxt1, 2, or 3	Input	Optional arguments showing context to be saved in the CRTX if the allocation request fails because of unavailable resources. If the requested resource is not available, and the caller has supplied a callback routine in the CRTX, the **Context 1, 2, 3** arguments will be saved in the CRTX$Q_Context1, 2, or 3

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.

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$L_ITEM_CNT	Number of items to be allocated. When requesting map registers, this value in this field should include two extra map registers to be allocated and loaded as guard pages to prevent runaway transfers.
CRCTX$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$L_SPINLOCK, raising IPL to IPL$_IOLOCK11 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 releases the spin lock indicated by CRAB$L_SPINLOCK.

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).

Prototype

int ioc$alloc_crab (int itemcnt, int gran, CRAB **crab_p)

Parameters

Name	Access	Description
itemcnt	Input	Number of items associated with the resource.
gran	Input	Requested allocation granularity associated with the resource.
crab_p	Input	Pointer to the Address of a cell to which IOC$ALLOC_CRAB returns the address of the allocated CRAB.

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.

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$L_SPINLOCK	Address of dynamic spin lock used to synchronize access to this CRAB. Currently, CRAB spin locks are obtained at IPL$_IOLOCK11.

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).

Prototype

int ioc$alloc_crctx (CRAB *crab, CRCTX **crctx_p, [int flck])

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.

Parameters

Name	Access	Description
crab	Input	Pointer to the address of CRAB that describes the counted resource. For adapters that supply a counted resource, such as map registers, ADP$L_CRAB often contains this address.
crctx_p	Input	Pointer to the address of a location in which IOC$ALLOC_CRCTX places the address of the allocated CRCTX.
flck	Input	Optional longword fork lock index. If this argument is not supplied, the routine defaults to spl$c_iolock8 saved in crctx$b_flck.

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$B_FLCK	Contents of **flck_index** argument if it is supplied, otherwise it defaults to IPL$C_IOLOCK8

IOC$ALLOCATE_CRAM

Allocates a controller register access mailbox.

Prototype

int ioc$allocate_cram (CRAM **cram_p, IDB *idb, UCB *ucb, ADP *adp)

Parameters

Name	Access	Description
cram	Input	Pointer to the address of the CRAM allocated by IOC$ALLOCATE_CRAM.
idb	Input	Pointer to the address of the IDB for the device.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
adp	Input	Pointer to the address of the 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.

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.

Prototype

int ioc$cancel_cnt_res (CRAB *crab, CRCTX *crctx, int resume)

Parameters

Name	Access	Description
crab	Input	Pointer to the address of CRAB that describes the counted resource. For adapters that supply a counted resource, such as map registers, ADP$L_CRAB often contains this address.
crctx	Input	Pointer to the address of CRCTX structure that describes the request for the counted resource.
resume	Input	Indication of whether the cancelled thread should be resumed. If true, IOC$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.

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.

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$_NORMAL status to its caller. Otherwise, after removing the CRCTX, it calls the driver's callback routine (CRCTX$L_CALLBACK), passing it the following values:

Location	Contents
R0, R21	SS$_CANCEL
R1, R16	Address of CRAB
R2, R17	Address of CRCTX
R3, R18	CRCTX$Q_FR3
R4, R19	CRCTX$Q_FR4
R5, R20	CRCTX$Q_FR5

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.

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.

Prototype

```
int ioc$cram_cmd (int cmdidx, int byteoffset, ADP *adp, CRAM *cram, uint64
    *iohandle)
```

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.

Parameters

Name	Access	Description
cmdidx	Input	IOC$CRAM_CMD uses this index to generate a mailbox command that is specific to the tightly-coupled interconnect that is to be the target of a request using this CRAM.
byteoffset	Input	Pointer to the byte offset of the field to be written or read from the base of device interface register (CSR) space. Calculation of the RBADR and MASK fields of the hardware mailbox depends on the addressing and masking mechanisms provided by the remote bus. The **byteoffset** argument is used by IOC$CRAM_CMD to calculate the RBADR.
adp	Input	Pointer to the 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.

Name	Access	Description
cram	Input	Pointer to the address of the 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.

Prototype

int ioc$cram_io (CRAM *cram)

Parameters

Name	Access	Description
cram	Input	Pointer to the address of CRAM associated with the hardware I/O mailbox transaction.

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.

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 successfully queues 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).

Prototype

int ioc$cram_queue (CRAM *cram)

Parameters

Name	Access	Description
cram	Input	Pointer to the address of the CRAM to be queued.

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.

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.

Prototype

int ioc$cram_wait (CRAM *cram)

Parameters

Name	Access	Description
cram	Input	Pointer to the address of CRAM associated with a previously-queued hardware I/O mailbox transaction.

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.

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.

Prototype

int ioc$dealloc_cnt_res (CRAB *crab, CRCTX *crctx)

Parameters

Name	Access	Description
crab	Input	Pointer to the address 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	Input	Pointer to the address of CRCTX structure that describes the request for the counted resource.

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.

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, if there are any waiting requests, queues a fork thread that will 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$DEALLOC_CNT_RES queues a fork thread that will attempt to restart any waiters for the resource.
6. When the fork thread eventually executes, it attempts to allocate the resource to any waiting CRCTX. If the fork thread succeeds in an allocation, the fork thread acquires the spinlock indicated by CRCTX$_FLCK, and invokes the callback routine indicated by CRCTX$L_CALLBACK.
7. If there are no waiters for the counted resource, IOC$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).

Prototype

int ioc$dealloc_crab (CRAB *crab)

Parameters

Name	Access	Description
crab	Input	Pointer to the address of the CRAB to be deallocated.

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.

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).

Prototype

int ioc$dealloc_crctx (CRCTX *crctx)

Parameters

Name	Access	Description
crctx	Input	Pointer to the address CRCTX to be deallocated.

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.

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.

Prototype

int ioc$deallocate_cram (CRAM *cram)

Parameters

Name	Access	Description
cram	Input	Pointer to the address of CRAM to be deallocated by IOC$DEALLOCATE_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.

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.

Prototype

int oc$kp_reqchan (KPB *kpb, int pri)

Parameters

Name	Access	Description
kpb	Input	Pointer to the address of the caller's kernel process block, 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.
pri	Input	Priority of the request for the controller channel.

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.

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.

   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.

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.

Prototype

int ioc$kp_wfikpch (KPB *kpb, int tmo, int newipl)

int ioc$kp_wfirlch (KPB *kpb, int tmo, int newipl)

Parameters

Name	Access	Description
kpb	Input	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	Input	Timeout value in seconds.
newipl	Input	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.

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.

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.

Prototype

```
int ioc$load_map (ADP *adp, CRCTX *crctx, PTE *svapte, int boff, void
    **dma_addr_p)
```

Parameters

Name	Access	Description
adp	Input	Pointer to the address of ADP for adapter which provides the map registers.
crctx	Input	Pointer to the 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	Input	Pointer to the system virtual address of the PTE for the first page to be used in the transfer.
boff	Input	Pointer to byte offset into the first page of the transfer buffer.
dma_address_ref	Input	Pointer to 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.

Context

IOC$LOAD_MAP conforms to the OpenVMS Alpha calling standard.

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

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, and Futurebus+ systems. It is not supported on XMI systems.

Prototype

```
int ioc$map_io (ADP *adp, int node, uint64 *physical_offset, int num_bytes, int
    attributes, uint64 *iohandle)
```

Parameters

Name	Access	Description
adp	Input	Address of bus ADP. Driver can get this from IDB$PS_ADP.
node	Input	Bus node number of device. Bus specific interpretation. Available to driver in CRB$L_ NODE (driver must be loaded with /NODE qualifier).
physical_offset	Input	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	Input	Number of bytes to be mapped. Expressed in terms of the bus/device without regard to the platform hardware addressing techniques.

Name	Access	Description
attributes	Input	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.

Name	Access	Description
iohandle	Input	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.

Return Values

SS$_NORMAL	The routine completed successfully. The address space is mapped. A 64 bit IOHANDLE is written to the caller's buffer.
SS$_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 DEC 2000 Model 150 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.

Prototype

int ioc$node_function (CRB *crb, int func)

Parameters

Name	Access	Description
crb	Input	Pointer to the address of the CRB.
func	Input	Pointer to the function to be effected for the bus node indicated by the **crb** argument.

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.

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 **func** 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

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.

Prototype

int int ioc$read_io (ADP *adp, uint64 *iohandle, int offset, int length, void *read_data)

Parameters

Name	Access	Description
adp	Input	Pointer to the address of bus ADP. Driver can get this from IDB$PS_ADP.
iohandle	Input	Pointer to a 64 bit IOHANDLE. The 64 bit IOHANDLE is obtained by calling the platform independent mapping routine IOC$MAP_IO.
offset	Input	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$MAP_IO. The offset is specified in terms of the device or bus without regard to any hardware address techniques.
length	Input	Length of field to be read or written. Should be 1 (byte), 2 (word), 3 (tribyte), 4 (longword) or 8 (quadword). Note that not all of these lengths are supported on all buses.
read_data	Input	Pointer to a data cell. For ioc$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.

Name	Access	Description
write_data	Input	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.

Return Values

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.

Prototype

int ioc$unmap_io (ADP *adp, uint64 *iohandle)

Parameters

Name	Access	Description
adp	Input	Address of bus ADP. Driver can get this from IDB$PS_ADP.
iohandle	Input	Pointer to a 64 bit IOHANDLE. The 64 bit IOHANDLE is obtained by calling the platform independent mapping routine IOC$MAP_IO.

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.

Prototype

int ioc$write_io (ADP *adp, uint64 *iohandle, int offset, int length, void *write_data)

Parameters

Name	Access	Description
adp	Input	Address of bus ADP. Driver can get this from IDB$PS_ADP.
iohandle	Input	Pointer to a 64 bit IOHANDLE. The 64 bit IOHANDLE is obtained by calling the platform independent mapping routine IOC$MAP_IO.
offset	Input	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$MAP_IO. The offset is specified in terms of the device or bus without regard to any hardware address techniques.
length	Input	Length of field to be read or written. Should be 1 (byte), 2 (word), 3 (tribyte), 4 (longword) or 8 (quadword). Note that not all of these lengths are supported on all buses.
read_data	Input	Pointer to a data cell. For IOC$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.

Name	Access	Description
write_data	Input	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.

Return Values

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.

Prototype

void ioc_std$altreqcom (int iost1, int iost2, CDRP *cdrp, IRP **irp_p, UCB **ucb_p)

Parameters

Name	Access	Description
iost1	Input	Pointer to the first longword of I/O status.
iost2	Input	Pointer to the second longword of I/O status.
cdrp	Input	Pointer to the class driver request packet.
irp_p	Input	Pointer to the address at which IOC_STD$ALTREQCOM writes the address of the I/O request packet.
ucb_p	Input	Pointer to the address at which IOC_STD$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.

IOC_STD$BROADCAST

Broadcasts the specified message to a given terminal.

Prototype

int ioc_std$broadcast (int msglen, void *msg_p, UCB *ucb)

Parameters

Name	Access	Description
msglen	Input	Pointer to the message length.
msg_p	Input	Pointer to the message.
ucb	Input	Pointer to the 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.

IOC_STD$CANCELIO

Conditionally marks a UCB so that its current I/O request will be canceled.

Prototype

void ioc_std$cancelio (int chan, IRP *irp, PCB *pcb, UCB *ucb)

Parameters

Name	Access	Description
chan	Input	Pointer to the channel index number.
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel. 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.

Parameter Fields

Field	Contents
irp->	
IRP$L_PID	Process identification of the process that queued the I/O request
IRP$L_CHAN	I/O request channel index number

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).

IOC_STD$CLONE_UCB

Copies a template UCB and links it to the appropriate DDB list.

Prototype

int ioc_std$clone_ucb (UCB *tmpl_ucb, UCB **new_ucb)

Parameters

Name	Access	Description
tmpl_ucb	Input	Pointer to the template unit control block.
new_ucb_p	Input	Pointer to the 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.

IOC_STD$COPY_UCB

Copies and initializes a template UCB and ORB.

Prototype

int ioc_std$copy_ucb (UCB *src_ucb, UCB **new_ucb)

Parameters

Name	Access	Description
src_ucb	Input	Pointer to the template unit control block.
new_ucb	Input	Pointer to the 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.

IOC_STD$CREDIT_UCB

Credits the UCB charges associated with a given UCB against the process identified by the contents of UCB$L_CPID.

Prototype

void ioc_std$credit_ucb (UCB *ucb)

Parameters

Name	Access	Description
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Context

A driver calls IOC_STD$CREDIT_UCB at IPL$_ASTDEL.

IOC_STD$CVT_DEVNAM

Converts a device name and unit number to a physical device name string.

Prototype

int ioc_std$cvt_devnam (int buflen, char *buf, int form, UCB *ucb, int32 *outlen_p)

Parameters

Name	Access	Description
buflen	Input	Pointer to the size of output buffer in bytes.
buf	Input	Pointer to the output buffer.
form	Input	Pointer to the name string formation mode. See Description section for more information.
ucb	Input	Pointer to the unit control block for device.
outlen_p	Input	Pointer to the 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

The **form** argument uses the 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)
4 (no GETDVI item code)	Path controller name for displays (same as –1 except no unit number is appended)

IOC_STD$CVTLOGPHY

Conditionally converts a logical block number to a physical disk address and stores the result in the I/O request packet.

Prototype

```
void ioc_std$cvtlogphy (int lbn, IRP *irp, UCB *ucb)
```

Parameters

Name	Access	Description
lbn	Input	Pointer to the logical block number to be converted.
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Context

A driver calls IOC_STD$CVTLOGPHY at fork IPL with the corresponding fork lock held in a multiprocessing system.

IOC_STD$DELETE_UCB

Deletes the specified UCB if its reference count is zero and UCB$V_DELETEUCB is set in UCB$L_STS.

Prototype

void ioc_std$delete_ucb (UCB *ucb)

Parameters

Name	Access	Description
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Context

A driver calls IOC_STD$DELETE_UCB with the I/O database locked for write access.

IOC_STD$DIAGBUFILL

Fills a diagnostic buffer if the original $QIO request specified such a buffer.

Prototype

void ioc_std$diagbufill (int driver_param, UCB *ucb)

Parameters

Name	Access	Description
driver_param	Input	Pointer to the parameter to be passed to the driver's register dumping routine. Typically, a driver supplies the address of a CRAM in this register.
ucb	Input	Pointer to the 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$L_IRP. IOC_STD$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.

Parameter Fields

Field	Contents
irp->	
IRP$L_STS	IRP$V_DIAGBUF set if a diagnostic buffer exists
IRP$L_DIAGBUF	Address of diagnostic buffer, if one is present

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.

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.

Prototype

void *ioc_std$filspt (UCB *ucb)

Parameters

Name	Access	Description
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel. 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.

IOC_STD$GETBYTE

Fetches a single byte of data from a user buffer.

Prototype

int ioc_std$getbyte (void *sva, UCB *ucb, void **sva_p)

Parameters

Name	Access	Description
sva	Input	Pointer to the 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	Input	Pointer to the Unit control block. IOC_STD$GETBYTE updates UCB$L_SVAPTE whenever a page boundary is crossed.
sva_p	Input	Pointer to the 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.

IOC_STD$INITBUFWIND

Initializes a single-page window into a user buffer.

Prototype

```
void *ioc_std$initbufwind (UCB *ucb)
```

Parameters

Name	Access	Description
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

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.

IOC_STD$INITIATE

Initiates the processing of the next I/O request for a device unit.

Prototype

void ioc_std$initiate (IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Parameter Fields

Field	Contents
irp-> read fields	
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-> read fields	
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.

Field	Contents
ucb-> write fields	
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.

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.

Prototype

int ioc_std$link_ucb (UCB *ucb)

Parameters

Name	Access	Description
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

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.

IOC_STD$MAPVBLK

Maps a virtual block to a logical block using a mapping window.

Prototype

```
int ioc_std$mapvblk (int vbn, int numbytes, WCB *wcb, IRP *irp, UCB *ucb, int32
    *lbn_p, int32 *notmapped_p, UCB **new_ucb_p)
```

Parameters

Name	Access	Description
vbn	Input	Pointer to the virtual block number.
numbytes	Input	Pointer to the number of bytes to map.
wcb	Input	Pointer to the window control block.
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the unit control block.
lbn_p	Output	Pointer to the address at which IOC_STD$MAPVBLK writes the logical block number of the first block it maps.
notmapped_p	Output	Pointer to the address at which IOC_STD$MAPVBLK writes the number of unmapped bytes.
new_ucb_p	Output	Pointer to the 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.

IOC_STD$MNTVER

Assists a driver with mount verification.

Prototype

void ioc_std$mntver (IRP *irp, UCB *ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet. 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	Input	Pointer to the unit control block of the device assigned to the process I/O channel.

Context

IOC_STD$MNTVER is called at fork IPL with the corresponding fork lock held in a multiprocessing system.

IOC_STD$MOVFRUSER, IOC_STD$MOVFRUSER2

Move data from a user buffer to an internal buffer.

Prototype

void *ioc_std$movfruser (void *sysbuf, int numbytes, UCB *ucb, void **sysbuf_p)

void *ioc_std$movfruser2 (void *sysbuf, int numbytes, UCB *ucb, void *sva, void **sysbuf_p)

Parameters

Name	Access	Description
sysbuf	Input	Pointer to the address of internal buffer.
numbytes	Input	Pointer to the number of bytes to move.
ucb	Input	Pointer to the unit control block.
sva	Input	Pointer to the system virtual address of the byte in the user buffer after the last byte moved (IOC_STD$MOVFRUSER2 only).
sysbuf_p	Output	Pointer to the system virtual address of the byte in the user buffer after the last byte moved. IOC_STD$MOVFRUSER and IOC_STD$MOVFRUSER2 write this field.

Parameter Fields

Field	Contents
ucb->	
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).

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.

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.

IOC_STD$MOVTOUSER, IOC_STD$MOVTOUSER2

Move data from an internal buffer to a user buffer.

Prototype

void *ioc_std$movtouser (void *sysbuf, int numbytes, UCB *ucb, void **sysbuf_p)

void *ioc_std$movtouser2 (void *sysbuf, int numbytes, UCB *ucb, void *sva, void **sysbuf_p)

Parameters

Name	Access	Description
sysbuf	Input	Pointer to the address of internal buffer.
numbytes	Input	Pointer to the number of bytes to move.
ucb	Input	Pointer to the unit control block.
sva	Input	Pointer to the system virtual address of the byte in the user buffer after the last byte moved (IOC_STD$MOVTOUSER2 only).
sysbuf_p	Output	Pointer to the system virtual address of the byte in the user buffer after the last byte moved. IOC_STD$MOVTOUSER and IOC_STD$MOVTOUSER2 write this field.

Parameter Fields

Field	Contents
ucb->	
UCB$I_SVAPTE	System virtual address of PTE that maps the first page of the user buffer.
UCB$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).

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 19 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.

IOC_STD$PARSDEVNAM

Parses a device name string, checking its syntax and extracting the node name, allocation class number, and unit number.

Prototype

int ioc_std$parsdevnam (int devnamlen, char *devnam, int flags, int32 *unit_p, int32 *scslen_p, int32 *devnamlen_p, char **devnam_p, int32 *flags_p)

Parameters

Name	Access	Description
devnamlen	Input	Pointer to the size of the name string.
devnam	Input	Pointer to the name string.
flags	Input	Pointer to the flags.
unit_p	Output	Pointer to the address at which IOC_STD$PARSDEVNAM writes an integer representing the unit number.
scslen_p	Output	Pointer to the address at which IOC_STD$PARSDEVNAM writes an integer representing either the length of the SCS node name, the allocation class number, or the device type code.
devnamlen_p	Output	Pointer to the address at which IOC_STD$PARSDEVNAM writes an integer representing the size of the name string.
devnam_p	Output	Pointer to the address at which IOC_STD$PARSDEVNAM writes the address of the name string.
flags_p	Output	Pointer to the 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.

IOC_STD$POST_IRP

Inserts an I/O request packet in a CPU-specific I/O postprocessing queue.

Prototype

void ioc_std$post_irp (IRP *irp)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.

Context

Mount verification processing calls IOC_STD$POST_IRP at or above IPL$_ASTDEL.

IOC_STD$PTETOPFN

Returns a page frame number (PFN) from a page-table entry (PTE) that has already been determined to be invalid.

Prototype

int ioc_std$ptetopfn (PTE *pte)

Parameters

Name	Access	Description
pte	Input	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.

IOC_STD$QNXTSEG1

Queues the next segment of a virtual I/O request that did not map to a single contiguous I/O request.

Prototype

```
void ioc_std$qnxtseg1 (int vbn, int bcnt, WCB *wcb, IRP *irp, PCB *pcb, UCB
    *ucb, UCB **ucb_p)
```

Parameters

Name	Access	Description
vbn	Output	Pointer to the virtual block number of the start of the next segment.
bcnt	Output	Pointer to the required byte count of next segment.
wcb	Output	Pointer to the window control block.
irp	Output	Pointer to the I/O request packet.
pcb	Output	Pointer to the process control block.
ucb	Output	Pointer to the unit control block.
ucb_p	Input	Pointer to the 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.

IOC_STD$PRIMITIVE_REQCHANH,REQCHANL

Request a controller's data channel and, if unavailable, place process in channel wait queue.

Prototype

int ioc_std$primitive_reqchanh (IRP *irp, UCB *ucb, IDB **idb_p)

int ioc_std$primitive_reqchanl (IRP *irp, UCB *ucb, IDB **idb_p)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
ucb	Input	Pointer to the Unit control block. 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	Output	Pointer to the address of location in which IOC_STD$REQPCHANH and IOC_STD$REQPCHANL write the address of the interrupt dispatch block (IDB).

Parameter Fields

Field	Contents
ucb->	
UCB$L_FPC	Procedure value of fork routine to be executed when the channel is granted if the channel cannot be granted immediately
UCB$L_CRB	Address of controller request block (CRB). IOC_STD$REQPCHANH and IOC_STD$REQPCHANL access the following CRB fields:
CRB fields	
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.

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.

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.

Prototype

void ioc_std$primitive_wfirlch (IRP *irp, int64 fr4, UCB *ucb, int tmo, int restore_ipl)

void ioc_std$primitive_wfirlch (IRP *irp, int64 fr4, UCB *ucb, int tmo, int restore_ipl)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
fr4	Input	Pointer to the parameter to be passed to the interrupt service routine or timeout handling routine.
ucb	Input	Pointer to the unit control block.
tmo	Input	Pointer to the timeout value in seconds.
restore_ipl	Input	Pointer to the 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.

Parameter Fields

Field	Contents
ucb-> read fields	
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$B_FLCK	Fork lock index.
ucb-> write fields	
UCB$L_DUETIM	Sum of timeout value and EXE$GL_ABSTIM
UCB$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.

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.

IOC_STD$RELCHAN

Releases device ownership of all controller data channels.

Prototype

void ioc_std$relchan (UCB *ucb)

Parameters

Name	Access	Description
ucb	Input	Pointer to the 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 CRB fields.

Parameter Fields

Field	Contents
crb->	
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.

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.

Prototype

void ioc_std$reqcom (int iost1, int iost2, UCB *ucb)

Parameters

Name	Access	Description
iost1	Input	Pointer to the first longword of I/O status.
iost2	Input	Pointer to the second longword of I/O status.
ucb	Input	Pointer to the unit control block.

Parameter Fields

Field	Contents
ucb->	
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.
UCB$L_OPCNT	Unit operations count. IOC_STD$REQCOM increases this field.
UCB$L_STS	If error logging is in progress (that is, UCB$V_ERLOGIP is set), IOC_STD$REQCOM writes the following fields in the error message buffer. IOC_STD$REQCOM then clears UCB$V_BSY and UCB$V_ERLOGIP.

Field	Contents
Error Message Buffer Fields	
EMB$L_DV_STS	UCB$L_STS.
EMB$L_DV_ERTCNT	UCB$L_ERTCNT.
EMB$L_DV_ERTCNT+1	UCB$L_ERTMAX.
EMB$Q_DV_IOSB	Quadword of I/O status.

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.

IOC_STD$SEARCHDEV

Searches the I/O database for a specific physical device.

Prototype

int ioc_std$searchdev (void *descr_p, UCB **ucb_p, DDB **ddb_p, SB **sb_p)

Parameters

Name	Access	Description
descr_p	Input	Pointer to the descriptor of device logical name.
ucb_p	Output	Pointer to the address at which IOC_STD$SEARCHDEV writes the unit control block (UCB) address.
ddb_p	Output	Pointer to the address at which IOC_STD$SEARCHDEV writes the device data block (DDB) address.
sb_p	Output	Pointer to the 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.

IOC_STD$SEARCHINT

Searches the I/O database for the specified device, using specified search rules.

Prototype

```
int ioc_std$searchint (int unit, int scslen, int devnamlen, char *devnam, int flags,
    UCB **ucb_p, DDB **ddb_p, SB **sb_p, void **lock_val_p)
```

Parameters

Name	Access	Description
unit	Input	Pointer to the unit number.
scslen	Input	Pointer to the integer representing either the length of the SCS node name, the allocation class number, or the device type code.
devnamlen	Input	Pointer to the size of the name string.
devnam	Input	Pointer to the name string.
flags	Input	Pointer to the flags.
ucb_p	Output	Pointer to the address at which IOC_STD$SEARCHINT writes the UCB address.
ddb_p	Output	Pointer to the address at which IOC_STD$SEARCHINT writes the DDB address.
sb_p	Output	Pointer to the address at which IOC_STD$SEARCHINT writes the system block (SB) address.
lock_val_p	Output	Pointer to the address at which IOC_STD$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.

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.

Prototype

int ioc_std$sensedisk (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet for the current I/O request.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process-I/O channel specified as an argument to the $QIO request.
ccb	Input	Pointer to the 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.

IOC_STD$SEVER_UCB

Removes the specified UCB from the UCB list of the device data block identified within the specified UCB.

Prototype

```
void ioc_std$sever_ucb (UCB *ucb)
```

Parameters

Name	Access	Description
ucb	Input	Pointer to the unit control block.

Context

A driver calls IOC_STD$SEVER_UCB with the I/O database locked for write access.

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.

Prototype

int ioc_std$simreqcom (int32 iosb[2], int pri, int efn, int32 iost[2] ACB *acb, int acmode)

Parameters

Name	Access	Description
iosb	Input	Pointer to the 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	Input	Pointer to the 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	Input	Pointer to the common or local event flag to be set. If this parameter contains –1, no event flag is set.
iost	Input	Pointer to the 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).

Name	Access	Description
acb	Input	Pointer to the 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	Input	Pointer to the 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.

IOC_STD$THREADCRB

Threads a controller request block (CRB) onto the CRB timeout queue chain headed by IOC$GL_CRBTMOUT.

Prototype

void ioc_std$threadcrb (CRB *crb)

Parameters

Name	Access	Description
crb	Input	Pointer to the controller request block.

Context

Mount verification processing calls IOC_STD$THREADCRB at or above IPL$_ASTDEL.

MMG_STD$IOLOCK

Locks process pages in memory.

Prototype

int mmg_std$iolock (void *buf, int bufsiz, int is_read, PCB *pcb, void **svapte_p)

Parameters

Name	Access	Description
buf	Input	Pointer to the buffer.
bufsize	Input	Pointer to the size of output buffer in bytes.
is_read	Input	Pointer to the transfer direction indicator, as follows: 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	Input	Pointer to the process control block.
svapte_p	Input	Pointer to the 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.

MMG_STD$UNLOCK

Unlocks process pages previously locked for a direct-I/O operation.

Prototype

void mmg_std$unlock (int npages, void *svapte)

Parameters

Name	Access	Description
npages	Input	Pointer to number of buffer pages to unlock.
svapte	Input	Pointer to 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.

MT_STD$CHECK_ACCESS

Checks access rights for magtape control write functions.

Prototype

int mt_std$check_access (IRP *irp, PCB *pcb, UCB *ucb, CCB *ccb)

Parameters

Name	Access	Description
irp	Input	Pointer to the I/O request packet.
pcb	Input	Pointer to the process control block of the current process.
ucb	Input	Pointer to the unit control block of the device assigned to the process I/O channel.
ccb	Input	Pointer to the channel control block for 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.

SCH_STD$IOLOCKR

Locks the I/O database mutex on behalf of its caller for read access.

Prototype

MUTEX *sch_std$iolockr (PCB *pcb)

Parameters

Name	Access	Description
pcb	Input	Pointer to the process control block of the current process.

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.

SCH_STD$IOLOCKW

Locks the I/O database mutex on behalf of its caller for write access.

Prototype

MUTEX *sch_std$iolockw (PCB *pcb)

Parameters

Name	Access	Description
pcb	Input	Pointer to the process control block of the current process.

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.

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.

Prototype

```
void sch_std$iounlock (PCB *pcb)
```

Parameters

Name	Access	Description
pcb	Input	Pointer to the process control block of the current process.

Context

SCH_STD$IOUNLOCK must be called below IPL$_SCHED. It returns to its caller at its caller's IPL.

20

C Driver Macros

This chapter describes the C macros that can be used by OpenVMS Alpha device drivers. These macros are defined in the `vms_macros.h` and `vms_drivers.h` header files. Any definitions of data types, function prototypes, and other macros that these macros require are also included in the these header files.

DEVICE_LOCK

Use to acquire a device spinlock and to optionally save the original IPL.

Format

device_lock (lockaddr, raise_ipl, savipl_p)

Parameters

Name	Access	Description
lockaddr	Input	Pointer to the device spinlock structure of type SPL.
raise_ipl	Input	Either the integer value RAISE_IPL or NORAISE_IPL. The symbol RAISE_IPL is defined to be 1 and the symbol NORAISE_IPL is defined to be 0 by the `vms_drivers.h` file. If `raise_ipl` is equal to RAISE_IPL, then the IPL is set using the value in the spinlock structure. Caution should be exercised not to use the constant NOLOWER_IPL for the raise_ipl parameter. If NOLOWER_IPL is used erroneously, the effect is that IPL will be raised. However, the effect of using values other than RAISE_IPL or NORAISE_IPL for the `raise_ipl` parameter should be considered as unpredictable. Remember that IPL is raised (or held) when acquiring a spin lock and lowered (or held) on release.
savipl_p	Output	Pointer to a 32-bit integer passed by reference in which the original IPL is returned. If the address of this parameter is NOSAVE_IPL, then the current IPL is not returned. The symbol NOSAVE_IPL is defined to be a null pointer, that is, ((int *) 0), by the `vms_drivers.h file`.

Defined by:

```
#include <vms_drivers.h>.
```

DEVICE_UNLOCK

Use to either release or restore (that is, conditionally release) a device spinlock and to optionally set a new IPL.

Format

device_unlock (lockaddr, newipl, restore)

Parameters

Name	Access	Description
lockaddr	Input	Pointer to the the device spinlock structure of type SPL.
newipl	Input	The integer value of the desired new IPL or the value NOLOWER_IPL if the IPL should be left unchanged. The symbol NOLOWER_IPL is defined to be -1 by the vms_drivers.h file. Caution should be exercised not to use the constants RAISE_IPL nor NORAISE_IPL instead of NOLOWER_IPL for the newipl parameter. If either of these are used erroneously in place of NOLOWER_IPL the effect is that IPL will be set to either 0 or 1. Remember that IPL is raised (or held) when acquiring a spin lock and lowered (or held) on release.
restore	Input	Either the integer value SMP_RESTORE or SMP_RELEASE. If SMP_RELEASE is specified then the spinlock is unconditionally released by calling SMP_STD$RELEASEL, otherwise the spinlock is conditionally released by calling SMP_STD$RESTOREL. The symbol SMP_RESTORE is defined to be 1 and the symbol SMP_RELEASE is defined to be 0 by the vms_drivers.h file.

Defined by:

```
#include <vms_drivers.h>
```

DSBINT

Use to set processor IPL to specified value and save previous value. Usually used to raise IPL and paired with a subsequent use of the enbint macro to return to the previous IPL.

Format

dsbint (newipl,saved_ipl)

Parameters

Name	Access	Description
newipl	Input	The integer value of the desired new IPL.
saved_ipl	Output	An integer variable into which the previous IPL is written.

Defined by:

```
#include <vms_macros.h>
```

ENBINT

Use to set processor IPL to specified value. Usually used to return to the previous IPL obtained from an earlier use of the dsbint macro.

Format

enbint (newipl)

Parameters

Name	Access	Description
newipl	Input	The integer value of the desired new IPL.

Defined by:

```
#include <vms_macros.h>
```

FORK

Use to queue a specified fork routine with specified fork routine parameters. After the fork routine is queued, execution continues with the next statement following the fork macro.

Format

fork (fork_routine, fr3, fr4, fkb)

Parameters

Name	Access	Description
fork_routine	Input	The procedure value of the routine that is to be executed in a fork thread. This value is passed to the fork dispatcher via fkb→fkb$l_fpc.
fr3	Input	The 64-bit value to pass to the fork routine via fkb→fkb$q_fr3. This parameter is cast as a 64-bit integer.
fr4	Input	The 64-bit value to pass to the fork routine via fkb→fkb$q_fr4. This parameter is cast as a 64-bit integer.
fkb	Input	Pointer to the fork block. This parameter is cast as a pointer to an FKB.

Defined by:

```
#include <vms_drivers.h>
```

FORK_LOCK

Use to acquire a fork spin lock and to optionally save the original IPL.

Format

fork_lock (lockidx, savipl_p)

Parameters

Name	Access	Description
lockidx	Input	The integer value of the spin lock index.
savipl_p	Output	Pointer to the 32-bit integer in which the original IPL is returned. If the address of this parameter is NOSAVE_IPL, then the original IPL is not returned. The symbol NOSAVE_IPL is defined to be a null pointer, i.e. ((int *) 0), by the `vms_drivers.h` file.

Defined by:

```
#include <vms_drivers.h>
```

FORK_UNLOCK

Use to either release or restore (i.e. conditionally release) a fork spin lock and to optionally set a new IPL.

Format

fork_unlock (lockidx, newipl, restore)

Parameters

Name	Access	Description
lockidx	Input	The integer value of the spin lock index.
newipl	Input	The integer value of the desired new IPL or the value NOLOWER_IPL if the IPL should be left unchanged. The symbol NOLOWER_IPL is defined to be -1 by the vms_drivers.h file. Caution should be exercised not to use the constants RAISE_IPL nor NORAISE_IPL instead of NOLOWER_IPL for the newipl parameter. If either of these are used erroneously in place of NOLOWER_IPL the effect is that IPL will be set to either 0 or 1. Remember that IPL is raised (or held) when acquiring a spin lock and lowered (or held) on release.
restore	Input	Either the integer value SMP_RESTORE or SMP_RELEASE. If SMP_RELEASE is specified then the spin lock is unconditionally released by calling SMP_STD$RELEASE, otherwise the spin lock is conditionally released by calling SMP_STD$RESTORE. The symbol SMP_RESTORE is defined to be 1 and the symbol SMP_RELEASE is defined to be 0 by the vms_drivers.h file.

Defined by:

```
#include <vms_drivers.h>
```

FORK_WAIT

Use to queue a specified fork routine with specified fork routine parameters for delayed execution. After the fork routine is queued, execution continues with the next statement following the fork macro.

Format

fork_wait (fork_routine, fr3, fr4, fkb)

Parameters

Name	Access	Description
fork_routine	Input	Pointer to the procedure value of the routine that is to be executed in a fork thread. This value is passed to the fork dispatcher via fkb→fkb$l_fpc.
fr3	Input	The 64-bit value to pass to the fork routine via fkb→fkb$q_fr3. This parameter is cast as a 64-bit integer.
fr4	Input	The 64-bit value to pass to the fork routine via fkb→fkb$q_fr4. This parameter is cast as a 64-bit integer.
fkb	Input	Pointer to the fork block. This parameter is cast as a pointer to an FKB.

Defined by:

```
#include <vms_drivers.h>
```

IOFORK

Use to queue a fork routine with specified fork routine parameters. This macro is very similar to fork, except that the fork block is assumed to be a UCB and the ucb$v_tim bit is cleared before the fork routine is queued.

Format

iofork (fork_routine, fr3, fr4, ucb)

Parameters

Name	Access	Description
fork_routine	Input	Pointer to the procedure value of the routine that is to be executed in a fork thread. This value is passed to the fork dispatcher via ucb→ucb$l_fpc.
fr3	Input	The 64-bit value to pass to the fork routine via ucb→ucb$q_fr3. This parameter is cast as a 64-bit integer.
fr4	Input	The 64-bit value to pass to the fork routine via ucb→ucb$q_fr4. This parameter is cast as a 64-bit integer.
ucb	Input	Pointer to the unit control block. This parameter is cast as a pointer to a UCB.

Defined by:

#include vms_drivers.h

RFI (RESUME FROM INTERRUPT)

Use in an interrupt service routine to invoke the resume from interrupt routine that has been set up by either the `wfikpch` or `wfirlch` macros.

Note that it may be possible to eliminate the driver resume from interrupt routine (and thus the need to use the rfi macro) by moving some processing directly into the interrupt service routine and by resuming the driver in a fork routine. The driver fork routine would then be resumed from the interrupt service routine by:

```
ucb->ucb$v_tim = 0;
exe_std$queue_fork ( (FKB *) ucb );
```

Format

rfi (irp, fr4, ucb)

Parameters

Name	Access	Description
irp	Input	Pointer to an IRP type, but can be any value which is expected as the first parameter of the resume from interrupt routine.
fr4	Input	Pointer to value which is expected as the second parameter of the resume from interrupt routine.
ucb	Input	Pointer to a Unit Control Block and is the third parameter of the resume from interrupt routine. This parameter is cast as a pointer to an UCB.

Defined by:

```
#include <vms.drivers.h
```

SETIPL

Use to set processor IPL to specified value.

Format

setipl (newipl)

Parameter

Name	Access	Description
newipl	Input	Integer value of the desired new IPL.

Defined by:

```
#include <vms_macros.h>
```

SOFTINT

Use to requests a software initiated interrupt at the specified IPL level.

Format

softint (ipl)

Parameters

Name	Access	Description
ipl	Input	Pointer to the integer value of the IPL of the desired software initiated interrupt.

Defined by:

```
#include <vms_macros.h>
```

SYS_LOCK

Use to acquire a spinlock and raise IPL if necessary.

Format

sys_lock (lockname,change_ipl,saved_ipl)

Parameters

Name	Access	Description
lockname	Input	Spinlock name in uppercase, for example, MMG. The macro appends this name to the SPL$C_ prefix to form the spinlock index constant.
change_ipl	Input	Constant 0 if the caller knows that the current IPL is exactly the IPL required for this spinlock and that a change of IPL is not necessary, othrwise 1.
saved_ipl	Input	Pointer to an integer into which the original IPL is returned or is the constant 0 if the caller does not want to obtain the previous IPL.

Defined by:

```
#include <vms_macros.h>
```

SYS_UNLOCK

Use to release or restore a spinlock and optionally set IPL.

Format

sys_unlock (lockname,new_ipl,restore)

Parameters

Name	Access	Description
lockname	Input	Spinlock name in uppercase, for example, MMG. The macro appends this name to the SPL$C_ prefix to form the spinlock index constant.
new_ipl	Input	An integer value. If non-negative, the current IPL is set to this integer value after the spinlock is released or restored. If negative, the current IPL is not altered.
restore	Input	An integer value. If non-zero, the spinlock is restored to its previous state. If zero, the spinlock is unconditionally released.

Defined by:

```
#include <vms_macros.h>
```

WFIKPCH (Wait for Interrupt and Keep Channel)

Use to set up an interrupt resume routine and a device interrupt timeout routine without releasing the channel, that is, the CRB.

Format

wfikpch (resume_rout, tout_rout, irp, fr4, ucb, tmo, restore_ipl)

Parameters

Name	Access	Description
resume_rout	Input	Pointer to the procedure value of the resume from interrupt routine that is to be called by the interrupt service routine. This value is passed to the interrupt service routine via ucb→ucb$l_fpc.
tout_rout	Input	Pointer to the procedure value of the device interrupt timeout routine that may be called by EXE$TIMEOUT. This value is passed via ucb→ucb$ps_toutrout.
irp	Input	Pointer to an IRP type that is passed to the interrupt resume or timeout routine by way of ucb→ucb$q_fr3.
fr4	Input	A 64-bit value to pass to the resume from interrupt or timeout routine via ucb→ucb$q_fr4. This parameter is cast as a 64-bit integer.
ucb	Input	Pointer to a Unit Control Block. This parameter is cast as a pointer to an UCB.
tmo	Input	An integer specifying the timeout value in seconds.
restore_ipl	Input	An integer specifying the IPL to lower to prior to returning.

Defined by:

```
#include <vms_drivers.h>
```

WFIRLCH (Wait for Interrupt and Release Channel)

Use to set up an interrupt resume routine and a device interrupt timeout routine, and to release the channel, that is, CRB.

Format

wfirlch (resume_rout, tout_rout, irp, fr4, ucb, tmo, restore_ipl)

Parameters

Name	Access	Description
resume_rout	Input	Pointer to the procedure value of the resume from interrupt routine that is to be called by the interrupt service routine. This value is passed to the interrupt service routine via ucb→ucb$l_fpc.
tout_rout	Input	Pointer to the procedure value of the device interrupt timeout routine that may be called by EXE$TIMEOUT. This value is passed via ucb→ucb$ps_toutrout.
irp	Input	Pointer to an IRP type that is passed to the interrupt resume or timeout routine by way of ucb→ucb$q_fr3.
fr4	Input	A 64-bit value to pass to the resume from interrupt or timeout routine via ucb→ucb$q_fr4. This parameter is cast as a 64-bit integer.
ucb	Input	Pointer to a Unit Control Block. This parameter is cast as a pointer to an UCB.
tmo	Input	An integer specifying the timeout value in seconds.
restore_ipl	Input	An integer specifying the IPL to lower to prior to returning.

Defined by:

```
#include <vms_drivers.h>
```

Part VI

Appendixes

Part VI contains PCI and EISA system address maps, an example driver program, and an example ICBM program. It includes the following chapters:

- Appendix A illustrates OpenVMS Alpha system maps for platforms that support PCI and EISA devices.
- Appendix B contains a sample driver written in C.
- Appendix C contains a sample IOGEN configuration building module (ICBM).

A

OpenVMS Alpha System Address Maps

Figure A–1 AlphaServer 1000 Address Map as Seen by a PCI or an EISA Device

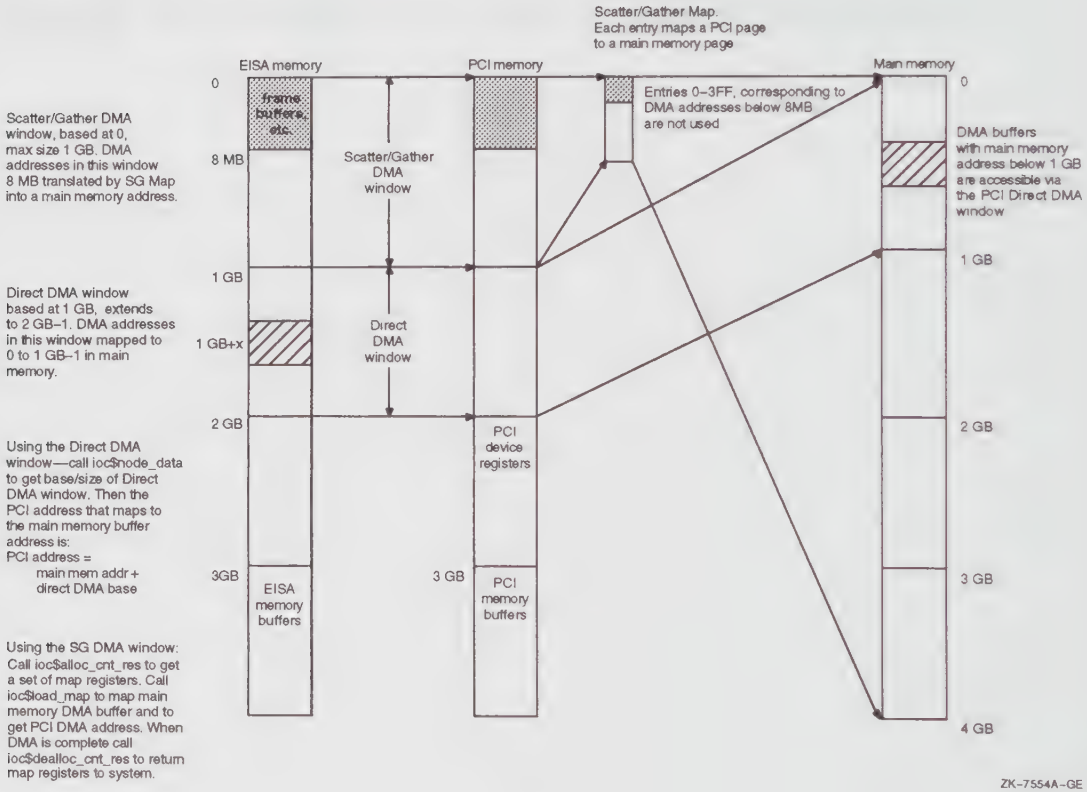


Figure A–2 AlphaServer 1000 Address Map as Seen by CPU

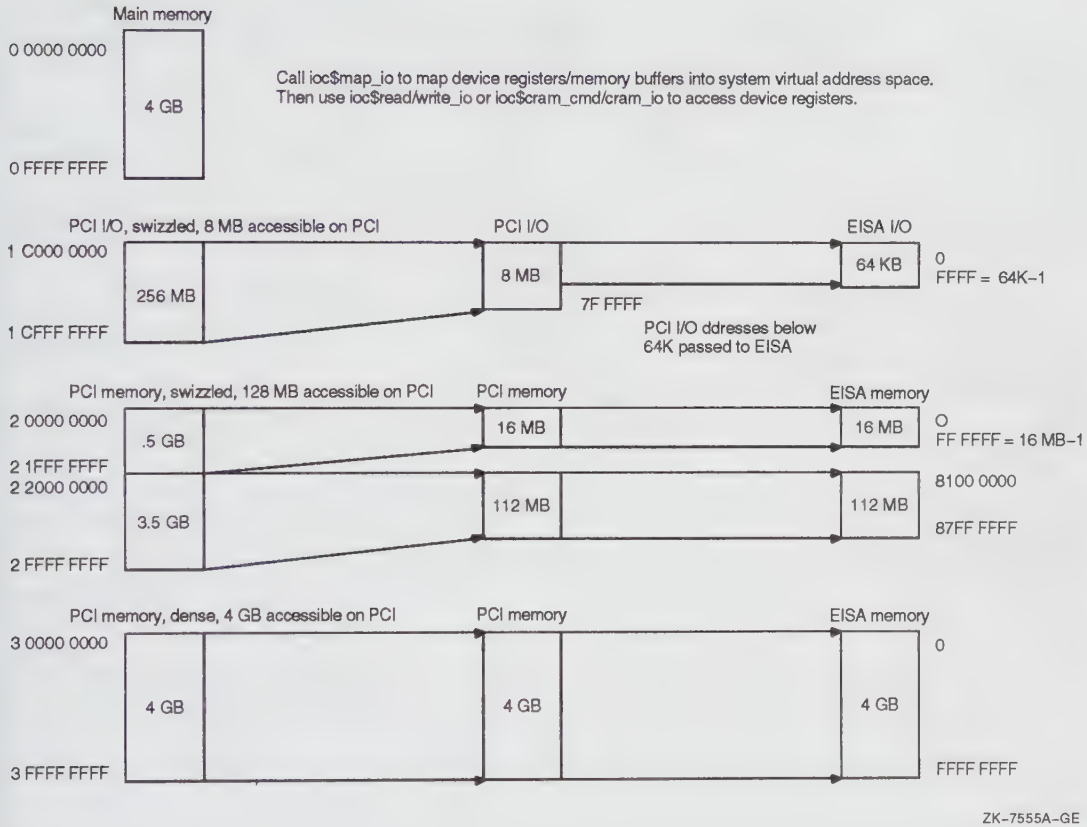


Figure A–3 AlphaStation 600 Address Map as Seen by a PCI or an EISA Device

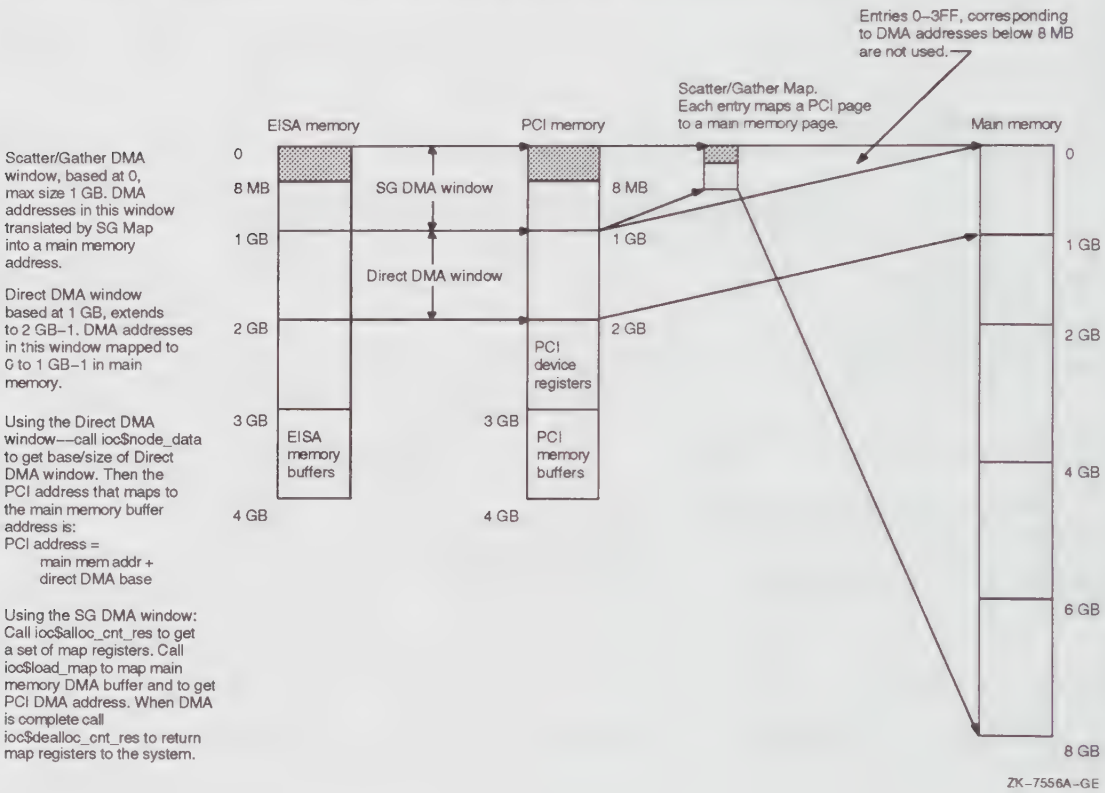


Figure A–4 AlphaServer 600 Platform Address Map as Seen by CPU

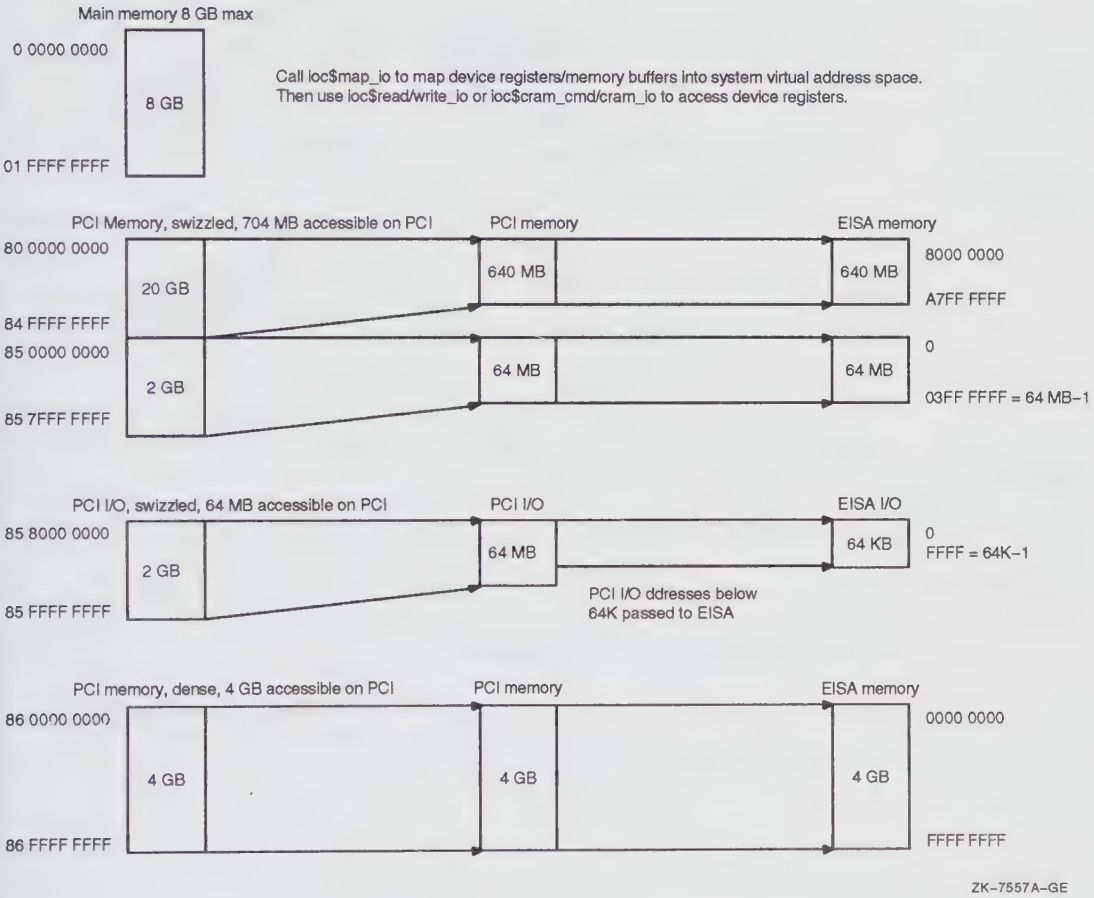


Figure A–5 AlphaServer 20000 Address Map as Seen by a PCI or an EISA Device

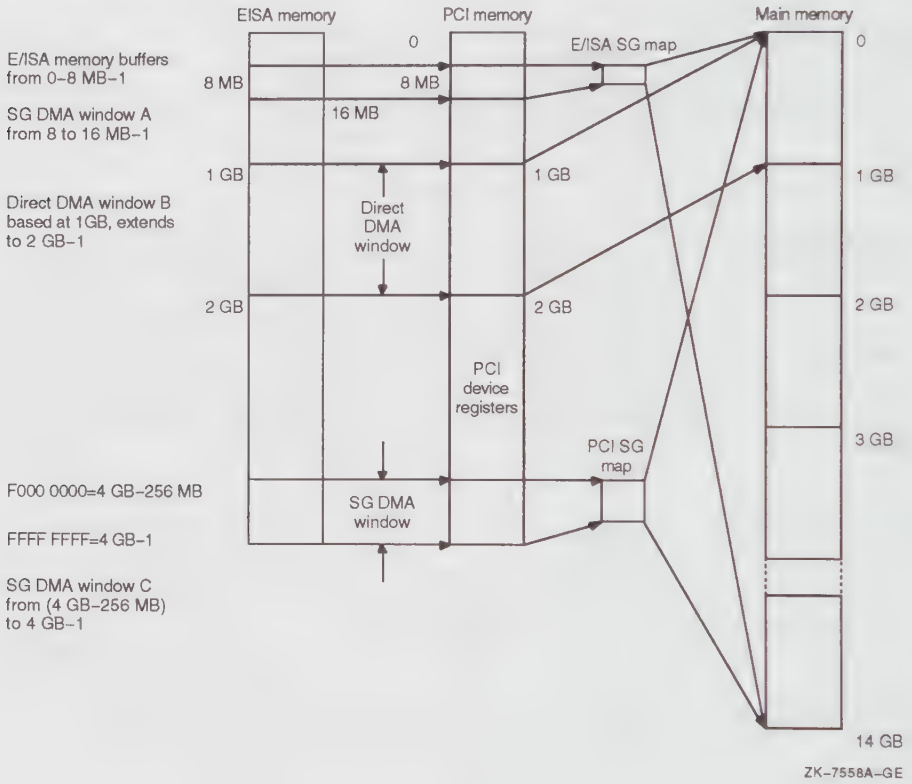


Figure A–6 AlphaServer 20000 Platform Address Map as Seen by CPU

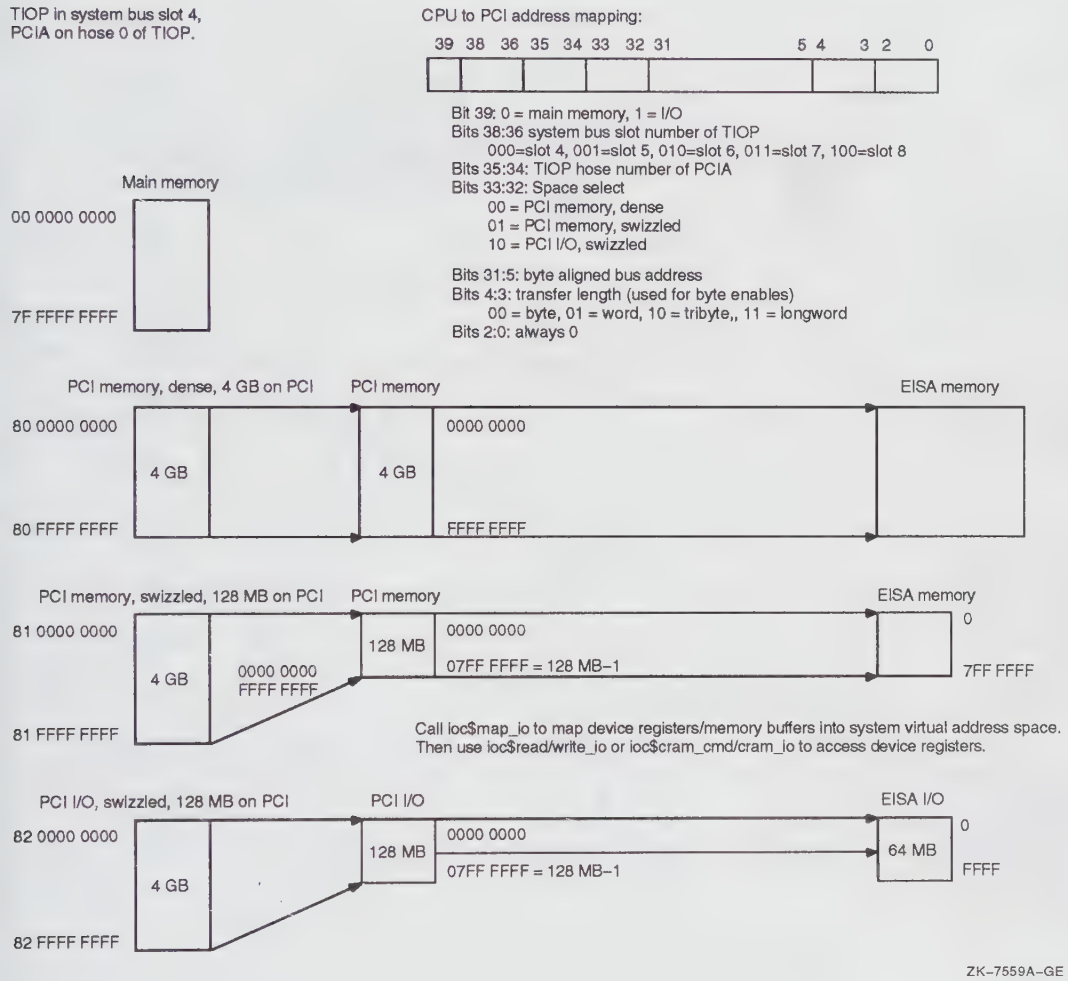


Figure A–7 AlphaServer 2100 Address Map as Seen by a PCI or an EISA Device

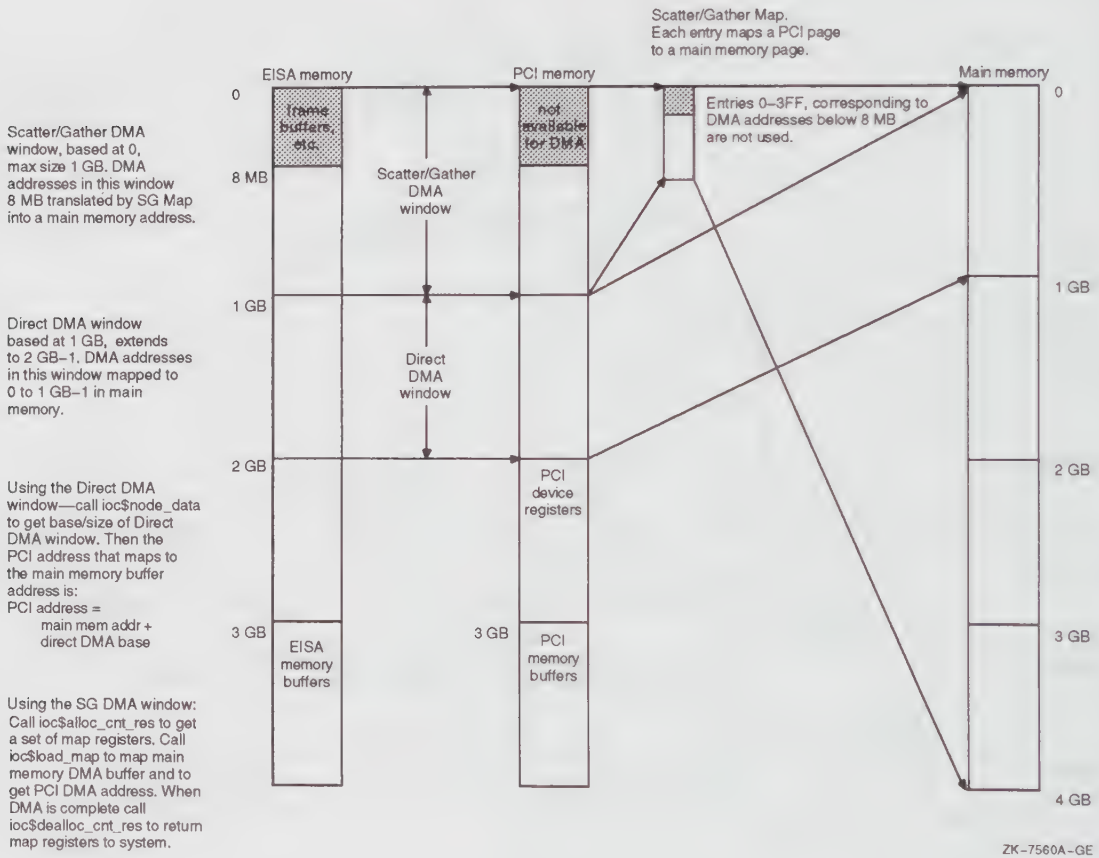


Figure A–8 AlphaServer 2100 Platform Address Map as Seen by CPU

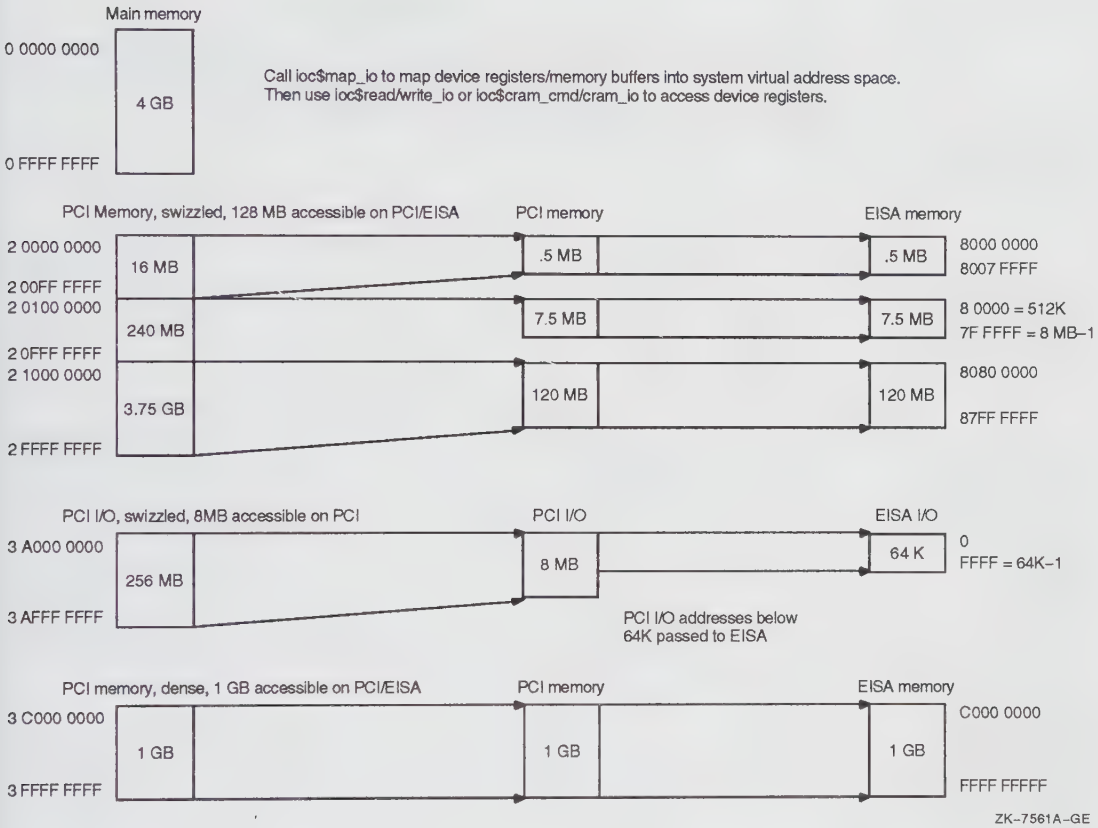


Figure A–9 AlphaServer 400, 200 Address Map as Seen by a PCI or an EISA Device

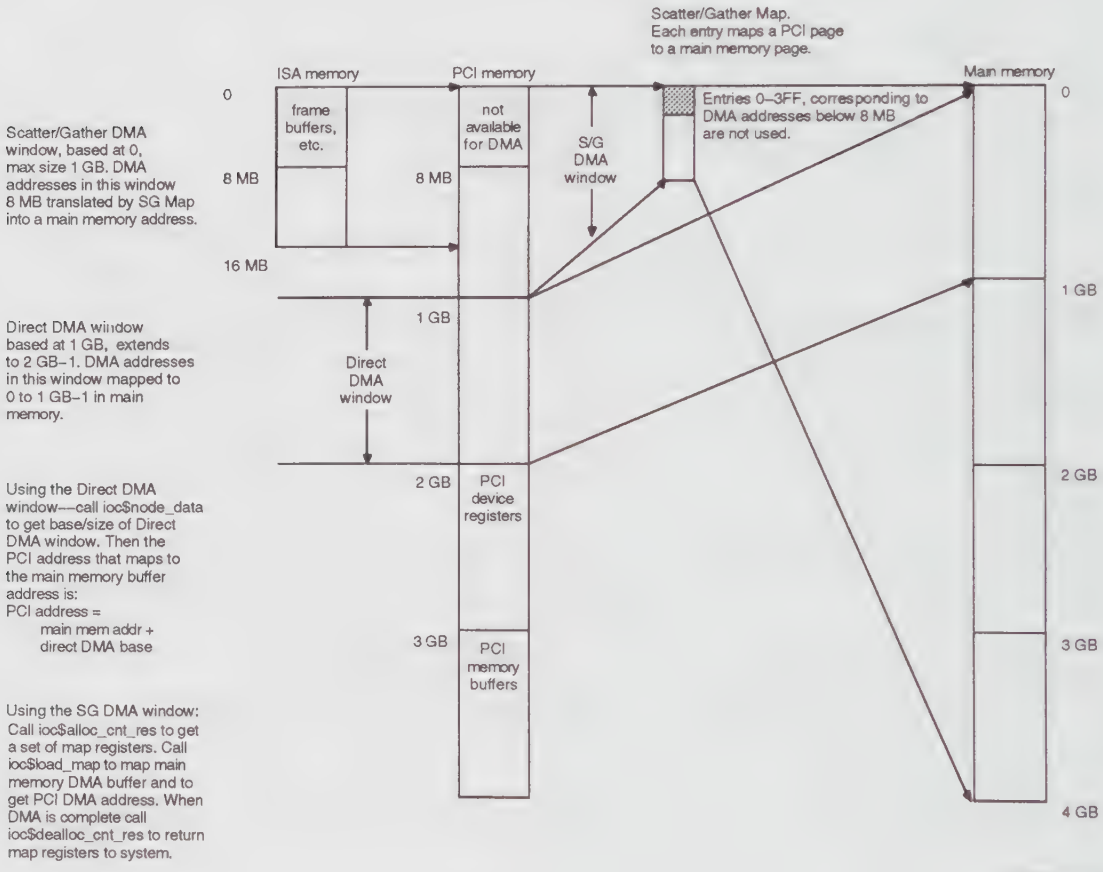


Figure A–10 AlphaServer 400, 200 Platform Address Map as Seen by CPU

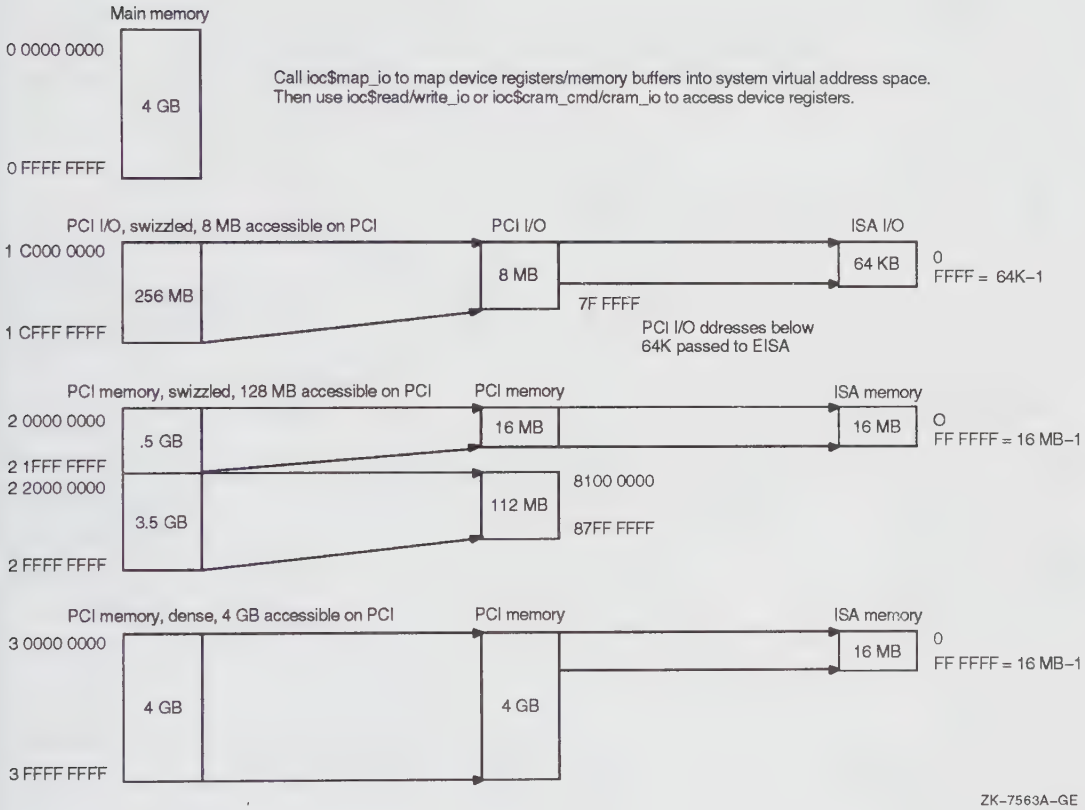


Figure A–11 DEC 2000 Platform Address Map as Seen by CPU

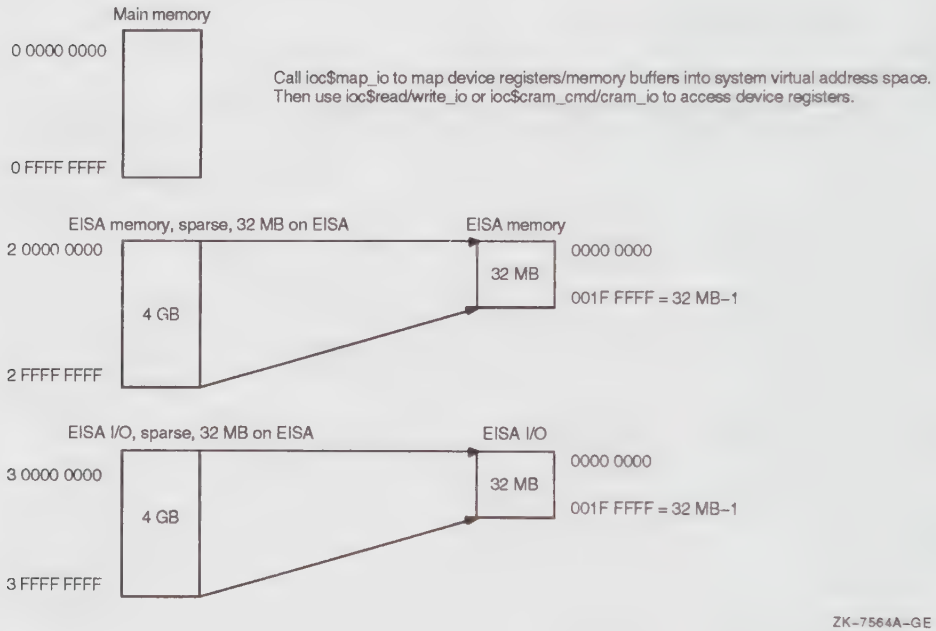
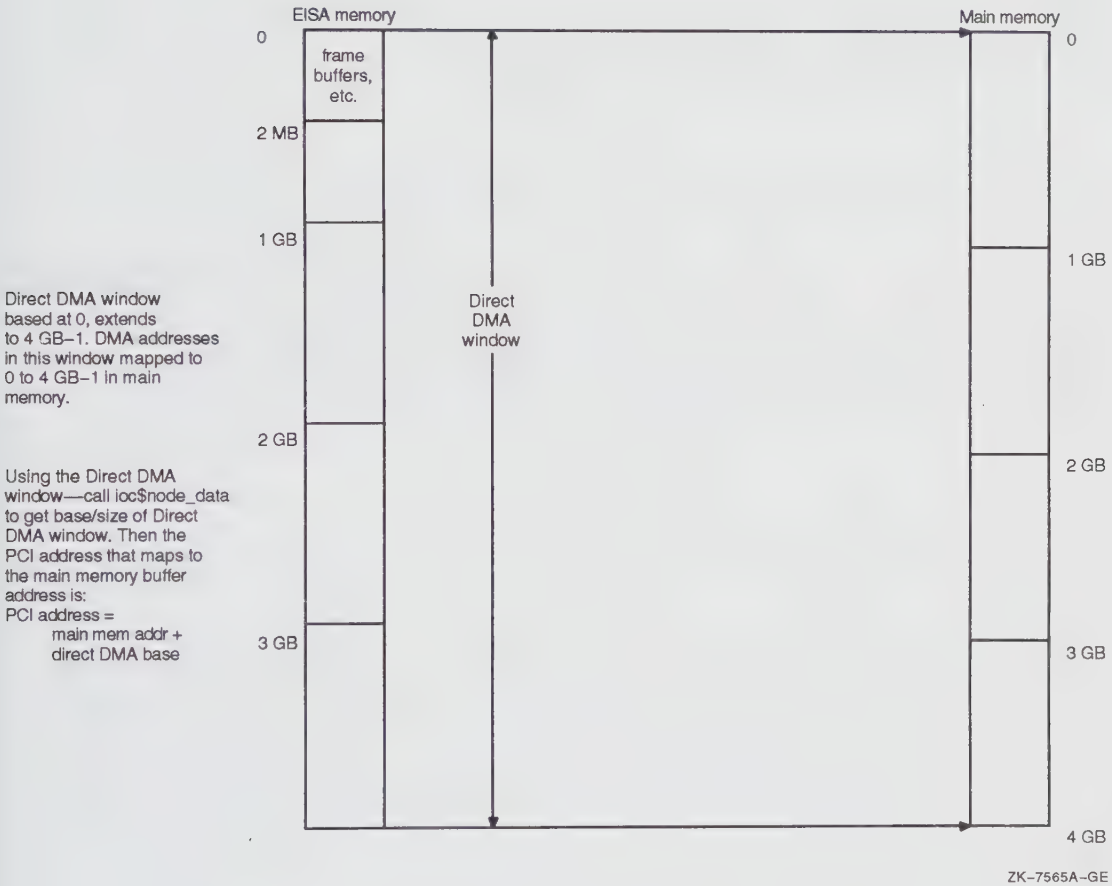


Figure A–12 DEC 2000 Address Map as Seen by an EISA Device



B

Sample Driver Written in C

This appendix contains a sample driver written in C and a command procedure for compiling and linking the driver.

B.1 LRDRIVER Example

The LRDRIVER is for the parallel output port of the VL82C106 Combo chip on an ISA option card for the DEC 2000 Model 300 Alpha system. You can obtain the most current version of this driver from the SYS$EXAMPLES directory.

```
#pragma module LRDRIVER "X-3"
/*
 ******************************************************************************
 *
 * Copyright © Digital Equipment Corporation, 1993, 1995 All Rights Reserved.
 * Unpublished rights reserved under the copyright laws of the United States.
 *
 * The software contained on this media is proprietary to and embodies the
 * confidential technology of Digital Equipment Corporation.  Possession, use,
 * duplication or dissemination of the software and media is authorized only
 * pursuant to a valid written license from Digital Equipment Corporation.
 *
 * RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure by the U.S.
 * Government is subject to restrictions as set forth in Subparagraph
 * (c)(1)(ii) of DFARS 252.227-7013, or in FAR 52.227-19, as applicable.
 *
 ******************************************************************************
 *
 *
 * FACILITY:
 *
 *      Example Device Driver for OpenVMS AXP
 *
 * ABSTRACT:
 *
 *      This is an example device driver for OpenVMS AXP Version V7.0 for the
 *      parallel printer port of the VL82C106 Combo chip.  This driver supports
 *      the VL82C106 either on the system bus or on an ISA option card.
 *
 *      The parallel printer port is a simple programmed I/O device.  There
```

```
 *      is a single control register (LPC), a status register (LPS), and a
 *      write data register (LWD).
 *
 *      This driver supports transfers from buffers in 64-bit virtual address
 *      spaces.
 *
 * AUTHOR:
 *
 *      OpenVMS Alpha Development Group
 *
 * REVISION HISTORY:
 *
 *      X-3     VMS001          OpenVMS Alpha Driver2   6-Jan-1996
 *              If call to exe_std$writechk fails abort the I/O.  The code
 *              assumed that the call aborted the I/O.  The routine claimed
 *              that is what it did.  The problem is that the code did not do
 *              what the routine description said it did.
 *
 *      X-2     VMS000          OpenVMS Alpha Drivers   29-Jun-1995
 *              This example driver is now the same source that is used to
 *              produce the SYS$LRDRIVER.EXE image that ships on the VMS kit.
 *              This driver supports transfers from buffers in a 64-bit
 *              virtual address space.
 *
 *      X-1     VMS000          OpenVMS Alpha Drivers    5-Nov-1993
 *              Initial version.
 *
 */

/* Define system data structure types and constants */
```

```
#include <bufiodef.h>           /* Define the packet header for a system */
                                /*   buffer for buffered I/O data */
#include <ccbdef.h>             /* Channel control block */
#include <crbdef.h>             /* Controller request block */
#include <cramdef.h>            /* Controller register access method */
#include <dcdef.h>              /* Device codes */
#include <ddbdef.h>             /* Device data block */
#include <ddtdef.h>             /* Driver dispatch table */
#include <devdef.h>             /* Device characteristics */
#include <dptdef.h>             /* Driver prologue table */
#include <fdtdef.h>             /* Function decision table */
#include <fkbdef.h>             /* Fork block */
#include <hwrpbdef.h>           /* Hardware restart parameter block */
#include <idbdef.h>             /* Interrupt data block */
#include <iocdef.h>             /* IOC constants */
#include <iodef.h>              /* I/O function codes */
#include <irpdef.h>             /* I/O request packet */
#include <ka0602def.h>          /* DEC 2000 Model 300 AXP specific defs  */
#include <lpdef.h>              /* Line printer definitions */
#include <orbdef.h>             /* Object rights block */
#include <pcbdef.h>             /* Process control block */
#include <msgdef.h>             /* System-wide mailbox message codes */
#include <ssdef.h>              /* System service status codes */
#include <stsdef.h>             /* Status value fields */
#include <ucbdef.h>             /* Unit control block */
#include <vecdef.h>             /* IDB interrupt transfer vector */

/* Define function prototypes for system routines */

#include <exe_routines.h>       /* Prototypes for exe$ and exe_std$ routines */
#include <ioc_routines.h>       /* Prototypes for ioc$ and ioc_std$ routines */
#include <sch_routines.h>       /* Prototypes for sch$ and sch_std$ routines */

/* Define various device driver macros */

#include <vms_drivers.h>        /* Device driver support macros, including */
                                /* table initialization macros and prototypes*/

/* Define the DEC C functions used by this driver */

#include <builtins.h>           /* OpenVMS AXP specific C builtin functions */
#include <string.h>             /* String routines provided by "kernel CRTL" */

#include "src$:lrdriver.h"      /* Fallback translation table */

/* Define constants specific to this driver */
```

```
enum {                                  /* Miscellaneous constants */
    FALSE               =   0,          /* True and False Flags */
    TRUE                =   1,
    DEVICE_IPL          =  21,          /*   Interrupt priority level of device */
    NUMBER_CRAMS        =   3,          /*   Number of CRAMs needed */
    LINES_PER_PAGE      =  66,          /*   Default paper size */
    DATA_EXPND_CUSHION =  32            /*   Extra room in system buffer for expansion */
};

enum {                                  /* Define various timeout constants */
    LR_WFI_TMO      =       15,         /*   Interrupt timeout value in seconds */
    LR_OFFLINE_TMO =       60,          /*   Initial interval between offline messages */
    ONE_HOUR        = (60*60)           /*   One hour in seconds */
};

enum {                                  /* Define names for some ASCII characters */
    CR  =               '\x0d',         /*   Carriage return character */
    DEL =               '\x7f',         /*   Delete */
    FF  =               '\x0c',         /*   Form Feed */
    HT  =               '\x09',         /*   Horizontal Tab */
    LF  =               '\x0a',         /*   Line feed character */
    SP  =               '\x20',         /*   Space */
    VT  =               '\x0b'          /*   Vertical Tab character */
};


/* Define the line printer port CSR offsets.
 *
 * Note: We have to do some special setup work because the Jensen built in
 * parallel port is on the system bus and is not byte-laned.  At the present
 * time other systems with built in parallel ports treat these as though they
 * live on the ISA bus.  The Unit Init routine figures how to correctly deal
 * with the built in controller and add on controllers.
 *
 * Note also that due to the byte-laned I/O space, data read from the ISA
 * LPS register (byte offset 1) must be shifted right 1 byte, and data read
 * from the LPC register (byte offset 2) must be shifted right 2 bytes.
 */

                                        /* Offsets for Jensen parallel port on system bus */
#define LR_JENSEN_LWD   0x3bc           /*   line printer port data write */
#define LR_JENSEN_LPS   0x3bd           /*   line printer port status */
#define LR_JENSEN_LCW   0x3be           /*   line printer port control write */

                                        /* Offsets for ISA space parallel port */
#define LR_LPT1_PORT    0x3bc           /*   ISA I/O address for LPT1 */
#define LR_LPT2_PORT    0x378           /*   ISA I/O address for LPT2 */
#define LR_LPT3_PORT    0x278           /*   ISA I/O address for LPT3 */
                                        /*   Actual register offset is LR_LPT2_PORT or */
                                        /*   LR_LPT2_PORT plus one of the following: */
#define LR_ISA_LWD        0x0           /*     line printer port data write */
#define LR_ISA_LPS        0x1           /*     line printer port status */
#define LR_ISA_LCW        0x2           /*     line printer port control write */
```

```
#define LR_LPT2_IRQ        7    /* Expected ISA IRQ for LPT2 */
#define LR_LPT3_IRQ        5    /* Expected ISA IRQ for LPT3 */

/* Line Printer Control Register
 * Mask values are defined for each of the control bits in the LPC.  This
 * driver always writes a new value to the LPC when a bit needs to be set.
 * A convenient way of doing this is to logically or together a subset of the
 * following masks to form the new LPC value.
 */
enum lpc_masks {
    LPC_M_STROBE    =  0x01,    /* Strobe data to printer */
    LPC_M_AUTO_FEED =  0x02,    /* Auto line feed enabled  */
    LPC_M_INIT_OFF  =  0x04,    /* Disable INIT signal */
    LPC_M_SELECT    =  0x08,    /* Select printer "on line" */
    LPC_M_IRQ_EN    =  0x10,    /* Interrupt enable */
    LPC_M_DIR_READ  =  0x20     /* Direction is read if set, else write */
};

/* Line Printer Status Register
 * Define a structure type with bit fields that corresponds to the status
 * bits.  This structure type facilitates the testing of these conditions.
 */
typedef struct _lps {
    unsigned int              : 2;      /* Reserved */
    unsigned int lps_irqp     : 1;      /* Interrupt pending */
    unsigned int lps_ok       : 1;      /* Ok status, i.e. no error */
    unsigned int lps_online   : 1;      /* Select on line */
    unsigned int lps_paperout : 1;      /* Paper empty */
    unsigned int lps_nak      : 1;      /* Not acknowledge */
    unsigned int lps_ready    : 1;      /* Ready, i.e. not busy */
} LPS;

/* Define a structure type for the carraige control information that
 * is returned from exe_std$carriage.  This information is returned in
 * the IRP at the longword that begins with irp->irp$b_carcon.
 */
typedef struct {
    uint8 prefix_count;     /* Number of prefix chars */
    char  prefix_char;      /* The prefix char, 0 if newline */
    uint8 suffix_count;     /* Number of suffix chars */
    char  suffix_char;      /* The suffix char, 0 if newline */
} CARCON;
```

```
/* Define a structure with the character formatting data in it.  This is passed
 * to character formatting code.
 */
typedef struct {
    int   buffer_space;     /* Size of system buffer in bytes */
    int   column_pos;       /* Column on page where character will go */
    int   cr_pend;          /* If printer is /CR flag indicating we held a CR */
    int   line_on_page;     /* Line numer we are currently on */
    int   page_length;      /* Length of page */
    int   page_width;       /* Width of papge in columns */
    int   total_bytes;      /* Numbers of bytes to output */
    int   total_lines;      /* Total lines printed for this I/O request */
    char  *sys_datap;       /* Pointer to current slot in system buffer */
} FMT_DATA;


/* Define Device-Dependent Unit Control Block with extensions for LR device */

typedef struct {
    UCB    ucb$r_ucb;                 /* Generic UCB */
    int    ucb$l_lr_msg_tmo;          /* Time out value for device offline msg */
    int    ucb$l_lr_oflcnt;           /* Offline time, print msg when reaches lr_msg_tmo */
    int    ucb$l_lr_cursor;           /* Current horizontal position */
    int    ucb$l_lr_lincnt;           /* Current line count on page */
    int    ucb$l_lr_cr_pend;          /* Pending CR flag */
    int    ucb$l_lr_jensen;           /* Unit is on system bus, not ISA option */
    int    ucb$l_lr_isa_io_address[2]/* ISA I/O address range */
    int    ucb$l_lr_isa_irq[2];       /* IRQ returned from ioc$node_data */
    CRAM   *ucb$ps_cram_lwd;          /* Line printer write data register */
    CRAM   *ucb$ps_cram_lps;          /* Line printer status register */
    CRAM   *ucb$ps_cram_lcw;          /* Line printer control register write */
    } LR_UCB;

/* Prototypes for driver routines defined in this module */

/* Driver table initialization routine */

    int  driver$init_tables ();

/* Device I/O database structure initialization routine */

    void lr$struc_init (CRB *crb, DDB *ddb, IDB *idb, ORB *orb, LR_UCB *ucb);

/* Device I/O database structure re-initialization routine */

    void lr$struc_reinit (CRB *crb, DDB *ddb, IDB *idb, ORB *orb, LR_UCB *ucb);

/* Unit initialization routine */

    int  lr$unit_init (IDB *idb, LR_UCB *ucb);

/* FDT routine for write functions */

    int  lr$write (IRP *irp, PCB *pcb, LR_UCB *ucb, CCB *ccb);
```

```
/* Output formatting routine */
    int  lr$format_char(LR_UCB *ucb, unsigned char out_char, FMT_DATA *fmt_data);
/* FDT routine for set mode and set characteristics functions */
    int  lr$setmode (IRP *irp, PCB *pcb, LR_UCB *ucb, CCB *ccb);
/* Start I/O routine */
    void lr$startio (IRP *irp, LR_UCB *ucb);
/* Local routine that sends the next character to the device */
    static int lr$send_char_dev (LR_UCB *ucb);
/* Interrupt service routine */
    void lr$interrupt (IDB *idb);
/* Driver fork routine entered when all I/O completed by interrupt service */
    void lr$iodone_fork (IRP *irp, void *not_used, LR_UCB *ucb);
/* Wait-for-interrupt timeout routine */
    void lr$wfi_timeout (IRP *irp, void *not_used, LR_UCB *ucb);
/* Periodic Check for Device Ready via Fork-wait mechanism */
    void lr$check_ready_fork (IRP *irp, void *not_used, LR_UCB *ucb);


/*
 * DRIVER$INIT_TABLES - Initialize Driver Tables
 *
 * Functional description:
 *
 *   This routine completes the initialization of the DPT, DDT, and FDT
 *   structures.  If a driver image contains a routine named DRIVER$INIT_TABLES
 *   then this routine is called once by the $LOAD_DRIVER service immediately
 *   after the driver image is loaded or reloaded and before any validity checks
 *   are performed on the DPT, DDT, and FDT.  A prototype version of these
 *   structures is built into this image at link time from the
 *   VMS$VOLATILE_PRIVATE_INTERFACES.OLB library.  Note that the device related
 *   data structures (e.g. DDB, UCB, etc.) have not yet been created when this
 *   routine is called.  Thus the actions of this routine must be confined to
 *   the initialization of the DPT, DDT, and FDT structures which are contained
 *   in the driver image.
 *
 * Calling convention:
 *
 *   status = driver$init_tables ();
 *
 * Input parameters:
 *
 *   None.
 *
```

```
 * Output parameters:
 *
 *   None.
 *
 * Return value:
 *
 *   status     If the status is not successful, then the driver image will
 *              be unloaded.  Note that the ini_* macros used below will
 *              result in a return from this routine with an error status if
 *              an initialization error is detected.
 *
 * Implicit inputs:
 *
 *  driver$dpt, driver$ddt, driver$fdt
 *              These are the externally defined names for the prototype
 *              DPT, DDT, and FDT structures that are linked into this driver.
 *
 * Environment:
 *
 *   Kernel mode, system context.
 */

int driver$init_tables ()  {

    /* Prototype driver DPT, DDT, and FDT will be pulled in from the
     * VMS$VOLATILE_PRIVATE_INTERFACES.OLB library at link time.
     */
    extern DPT driver$dpt;
    extern DDT driver$ddt;
    extern FDT driver$fdt;

    /* Finish initialization of the Driver Prologue Table (DPT) */

    ini_dpt_name        (&driver$dpt, "LRDRIVER");
    ini_dpt_adapt       (&driver$dpt, AT$_KA0602);
    ini_dpt_defunits    (&driver$dpt, 1);
    ini_dpt_ucbsize     (&driver$dpt, sizeof(LR_UCB));
    ini_dpt_struc_init  (&driver$dpt, lr$struc_init );
    ini_dpt_struc_reinit(&driver$dpt, lr$struc_reinit );
    ini_dpt_ucb_crams   (&driver$dpt, NUMBER_CRAMS);
    ini_dpt_end         (&driver$dpt);

    /* Finish initialization of the Driver Dispatch Table (DDT) */

    ini_ddt_unitinit    (&driver$ddt, lr$unit_init);
    ini_ddt_start       (&driver$ddt, lr$startio);
    ini_ddt_cancel      (&driver$ddt, ioc_std$cancelio);
    ini_ddt_end         (&driver$ddt);
```

```
    /* Finish initialization of the Function Decision Table (FDT)   */
    /*                                                              */
    /* The BUFFERED_64 indicates that this driver supports a 64-bit */
    /* virtual address in the QIO P1 parameter for that function.   */
    /* This driver, therefore, supports 64-bit user buffers in all  */
    /* of its I/O functions.                                        */

    ini_fdt_act (&driver$fdt, IO$_WRITELBLK, lr$write, BUFFERED_64);
    ini_fdt_act (&driver$fdt, IO$_WRITEPBLK, lr$write, BUFFERED_64);
    ini_fdt_act (&driver$fdt, IO$_WRITEVBLK, lr$write, BUFFERED_64);
    ini_fdt_act (&driver$fdt, IO$_SETMODE, lr$setmode, BUFFERED_64);
    ini_fdt_act (&driver$fdt, IO$_SETCHAR, lr$setmode, BUFFERED_64);
    ini_fdt_act (&driver$fdt, IO$_SENSEMODE, exe_std$sensemode, BUFFERED_64);
    ini_fdt_act (&driver$fdt, IO$_SENSECHAR, exe_std$sensemode, BUFFERED_64);
    ini_fdt_end (&driver$fdt);

    /* If we got this far then everything worked, so return success. */

    return SS$_NORMAL;
}


/*
 * LR$STRUC_INIT - Device Data Structure Initialization Routine
 *
 * Functional description:
 *
 *   This routine is called once for each unit by the $LOAD_DRIVER service
 *   after that UCB is created.  At the point of this call the UCB has not
 *   yet been fully linked into the I/O database.  This routine is responsible
 *   for filling in driver specific fields that in the I/O database structures
 *   that are passed as parameters to this routine.
 *
 *   This routine is responsible for filling in the fields that are not
 *   affected by a RELOAD of the driver image.  In contrast, the structure
 *   reinitialization routine is responsible for filling in the fields that
 *   need to be corrected when (and if) this driver image is reloaded.
 *
 *   After this routine is called for a new unit, then the reinitialization
 *   routine is called as well.  Then the $LOAD_DRIVER service completes the
 *   integration of these device specific structures into the I/O database.
 *
 *   Note that this routine must confine its actions to filling in these I/O
 *   database structures and may not attempt to initialize the hardware device.
 *   Initialization of the hardware device is the responsibility of the
 *   controller and unit initialization routines which are called some time
 *   later.
 *
 * Calling convention:
 *
 *   lr$struc_init (crb, ddb, idb, orb, ucb)
 *
 * Input parameters:
```

```
 *
 *   crb        Pointer to associated controller request block.
 *   ddb        Pointer to associated device data block.
 *   idb        Pointer to associated interrupt dispatch block.
 *   orb        Pointer to associated object rights block.
 *   ucb        Pointer to the unit control block that is to be initialized.
 *
 * Output parameters:
 *
 *   None.
 *
 * Return value:
 *
 *   None.
 *
 * Environment:
 *
 *   Kernel mode, system context, IPL may be as high as 31 and may not be
 *   altered.
 *
 */
void lr$struc_init (CRB *crb, DDB *ddb, IDB *idb, ORB *orb, LR_UCB *ucb) {

    /* Initialize the fork lock and device IPL fields */

    ucb->ucb$r_ucb.ucb$b_flck = SPL$C_IOLOCK8;
    ucb->ucb$r_ucb.ucb$b_dipl = DEVICE_IPL;

    /* Device Characteristics are :  Record oriented (REC), Available (AVL),
     * Carriage control device (CCL), Output device (ODV)
     */
    ucb->ucb$r_ucb.ucb$l_devchar = DEV$M_REC | DEV$M_AVL | DEV$M_CCL | DEV$M_ODV;

    /* Set to prefix device name with "node$", set device class, device type,
     *  and default buffer size.
     */
    ucb->ucb$r_ucb.ucb$l_devchar2 = DEV$M_NNM;
    ucb->ucb$r_ucb.ucb$b_devclass = DC$_LP;
    ucb->ucb$r_ucb.ucb$b_devtype = LP$_LP11;
    ucb->ucb$r_ucb.ucb$w_devbufsiz = 132;

    /* Lines per page in highest byte of ucb$l_devdepend and LP attributes
     * in lower three bytes.
     */
    ucb->ucb$r_ucb.ucb$l_devdepend = (LINES_PER_PAGE << 24) |
                                     LP$M_MECHFORM | LP$M_TRUNCATE;
    return;
}
```

```
/*
 * LR$STRUC_REINIT - Device Data Structure Re-Initialization Routine
 *
 * Functional description:
 *
 *   This routine is called once for each unit by the $LOAD_DRIVER service
 *   immediately after the structure initialization routine is called.
 *
 *   Additionally, this routine is called once for each unit by the $LOAD_DRIVER
 *   service when a driver image is RELOADED.  Thus, this routine is
 *   responsible for filling in the fields in the I/O database structures
 *   that point into this driver image.
 *
 *   Note that this routine must confine its actions to filling in these I/O
 *   database structures.
 *
 * Calling convention:
 *
 *   lr$struc_reinit (crb, ddb, idb, orb, ucb)
 *
 * Input parameters:
 *
 *   crb        Pointer to associated controller request block.
 *   ddb        Pointer to associated device data block.
 *   idb        Pointer to associated interrupt dispatch block.
 *   orb        Pointer to associated object rights block.
 *   ucb        Pointer to the unit control block that is to be initialized.
 *
 * Output parameters:
 *
 *   None.
 *
 * Return value:
 *
 *   None.
 *
 * Environment:
 *
 *   Kernel mode, system context, IPL may be as high as 31 and may not be
 *   altered.
 *
 */
void lr$struc_reinit (CRB *crb, DDB *ddb, IDB *idb, ORB *orb, LR_UCB *ucb) {

    extern DDT driver$ddt;

    /* Setup the pointer from our DDB in the I/O database to the driver
     * dispatch table that's within this driver image.
     */
    ddb->ddb$ps_ddt = &driver$ddt;
```

```
    /* Setup the procedure descriptor and code entry addresses in the VEC
     * portion of the CRB in the I/O database to point to the interrupt
     * service routine that's within this driver image.
     */
    dpt_store_isr (crb, lr$interrupt);

    return;
}


/*
 * LR$UNIT_INIT - Unit Initialization Routine
 *
 * Functional description:
 *
 *   This routine is called once for each unit by the $LOAD_DRIVER service
 *   after a new unit control block has been created, initialized, and
 *   fully integrated into the I/O database.
 *
 *   This routine is also called for each unit during power fail recovery.
 *
 *   It is the responsibility of this routine to bring unit "on line" and
 *   to make it ready to accept I/O requests.
 *
 * Calling convention:
 *
 *   status = lr$unit_init (idb, ucb)
 *
 * Input parameters:
 *
 *   idb        Pointer to associated interrupt dispatch block.
 *   ucb        Pointer to the unit control block that is to be initialized.
 *
 * Output parameters:
 *
 *   None.
 *
 * Return value:
 *
 *   status     SS$_NORMAL indicates that the unit was initialized successfully.
 *              SS$_IVADDR indicates that an unexpected ISA I/O address or IRQ
 *                         level was detected.
 *
 * Environment:
 *
 *   Kernel mode, system context, IPL 31.
 */

int lr$unit_init (IDB *idb, LR_UCB *ucb)  {

    extern uint64 EXE$GQ_SYSTYPE;
    static int jensen_combo_initialized = 0;          /* First unit is on system bus */
```

```
    CRAM *cram;
    ADP *adp;
    int isa_io_addr;            /* Slot I/O address if ISA option */
    int device_data;            /* Data from or for CRAM */
    int status;

#if defined DEBUG

    /* If a debug version of this driver is being built then invoke the loaded
     * system debugger.  This could either be the High Level Language System
     * Debugger, XDELTA, or nothing.
     */
    {
        extern void ini$brk (void);
        ini$brk ();
    }
#endif

    /* Set device initially offline (for error exits) and initialize other
     * UCB cells.
     */
    ucb->ucb$r_ucb.ucb$v_online = 0;

    ucb->ucb$l_lr_msg_tmo = LR_OFFLINE_TMO;

    /* This driver can service only a single unit per DDB and IDB.  Thus,
     * make the single unit the permanent owner of the IDB.  This facilitates
     * getting the UCB address in our interrupt service routine.
     */
    idb->idb$ps_owner = &(ucb->ucb$r_ucb);

    /* Initialize the three CRAMs that were requested in our DPT and allocated
     * before this unit initialization routine was called.
     */
    adp = ucb->ucb$r_ucb.ucb$ps_adp;            /* Pointer to our ADP */
    cram = ucb->ucb$r_ucb.ucb$ps_cram;          /* Pointer to first CRAM */

    /* If the system is a Jensen then we assume that the first port is
     * the VL82C106 on the system bus.  All subsequent units are in ISA
     * space.
     */
    if ( ! jensen_combo_initialized &&
         EXE$GQ_SYSTYPE == HWRPB_SYSTYPE$K_JENSEN) {

        jensen_combo_initialized = 1;   /* Unit on system bus initialized */
        ucb->ucb$l_lr_jensen = 1;       /* This unit is for VL82C106 on system bus */

        /* Initialize CRAM used to write the data register */

        cram->cram$v_der = 1;
        ucb->ucb$ps_cram_lwd = cram;
        ioc$cram_cmd (CRAMCMD$K_WTLONG32, LR_JENSEN_LWD, adp, cram, 0);

        /* Initialize CRAM used to read the status register */
```

```
        cram = cram->cram$l_flink;
        cram->cram$v_der = 1;
        ucb->ucb$ps_cram_lps = cram;
        ioc$cram_cmd (CRAMCMD$K_RDLONG32, LR_JENSEN_LPS, adp, cram, 0);

        /* Initialize CRAM used to write the control register */

        cram = cram->cram$l_flink;
        cram->cram$v_der = 1;
        ucb->ucb$ps_cram_lcw = cram;
        ioc$cram_cmd (CRAMCMD$K_WTLONG32, LR_JENSEN_LCW, adp, cram, 0);

    } else {                            /* This unit is ISA bus card */

        /* Get and validate the ISA IRQ */

        status = ioc$node_data (ucb->ucb$r_ucb.ucb$l_crb, IOC$K_EISA_IRQ,
                                &ucb->ucb$l_lr_isa_irq[0] );
        if ( ! $VMS_STATUS_SUCCESS(status) ) return status;

        /* Get and validate the ISA I/O address */

        status = ioc$node_data (ucb->ucb$r_ucb.ucb$l_crb, IOC$K_EISA_IO_PORT,
                                &ucb->ucb$l_lr_isa_io_address[0] );
        if ( ! $VMS_STATUS_SUCCESS(status) ) return status;

        isa_io_addr = ucb->ucb$l_lr_isa_io_address[0] & 0xfff;  /* Keep Address only */

        /* Initialize CRAM used to write the data register */

        cram->cram$v_der = 1;
        ucb->ucb$ps_cram_lwd = cram;
        ioc$cram_cmd (CRAMCMD$K_WTBYTE32, isa_io_addr+LR_ISA_LWD, adp, cram, 0);

        /* Initialize CRAM used to read the status register */

        cram = cram->cram$l_flink;
        cram->cram$v_der = 1;
        ucb->ucb$ps_cram_lps = cram;
        ioc$cram_cmd (CRAMCMD$K_RDBYTE32, isa_io_addr+LR_ISA_LPS, adp, cram, 0);

        /* Initialize CRAM used to write the control register */

        cram = cram->cram$l_flink;
        cram->cram$v_der = 1;
        ucb->ucb$ps_cram_lcw = cram;
        ioc$cram_cmd (CRAMCMD$K_WTBYTE32, isa_io_addr+LR_ISA_LCW, adp, cram, 0);

    }

    /* Enable interrupts */

    status = ioc$node_function (ucb->ucb$r_ucb.ucb$l_crb, IOC$K_ENABLE_INTR);
    if ( ! $VMS_STATUS_SUCCESS(status) ) return status;
```

```
    /* Set the INIT_OFF bit in the port control register.  The INIT signal is
     * asserted as long as INIT_OFF is clear.  Note byte-lane shift if ISA
     * option.
     */
    if (ucb->ucb$l_lr_jensen)
        device_data = LPC_M_INIT_OFF;
    else
        device_data = LPC_M_INIT_OFF << 16;

    ucb->ucb$ps_cram_lcw->cram$q_wdata = device_data;
    ioc$cram_io (ucb->ucb$ps_cram_lcw);

    /* Mark the device as "on line" and ready to accept I/O requests */

    ucb->ucb$r_ucb.ucb$v_online = 1;

    return SS$_NORMAL;
}


/*
 * LR$SETMODE - FDT Routine for Set Mode and Set Characteristics
 *
 * Functional description:
 *
 *   This routine is called by the FDT dispatcher in the $QIO system service
 *   to process set mode and set characteristics functions.  This FDT routine
 *   completes the I/O request without sending it to the driver start I/O
 *   routine. The user buffer address is contained in irp$q_qio_p1 ($QIO
 *   P1 parameter) on input and will be treated as a 64-bit address.
 *
 *   Since this is an upper-level FDT routine, this routine always returns
 *   the SS$_FDT_COMPL status.  The $QIO status that is to be returned to
 *   the caller of the $QIO system service is returned indirectly by the
 *   FDT completion routines (e. g. exe_std$abortio, exe_std$finishio) via
 *   the FDT context structure.
 *
 * Calling convention:
 *
 *   status = lr$setmode (irp, pcb, ucb, ccb)
 *
 * Input parameters:
 *
 *   irp        Pointer to I/O request packet
 *   pcb        Pointer process control block
 *   ucb        Pointer to unit control block
 *   ccb        Pointer to channel control block
 *
 * Output parameters:
 *
 *   None.
 *
 * Return value:
```

```
 *
 *   status     SS$_FDT_COMPL
 *
 * Environment:
 *
 *   Kernel mode, user process context, IPL 2.
 */

int  lr$setmode (IRP *irp, PCB *pcb, LR_UCB *ucb, CCB *ccb) {

    /* Define a structure that corresponds to the layout of the caller's
     * set mode or set characteristics buffer and declare a local pointer
     * to a structure of this type.
     */
    typedef struct {
        unsigned char  devclass;
        unsigned char  devtype;
        unsigned short devbufsiz;
        unsigned int   devdepend;
    } SETMODE_BUF;

    #pragma __required_pointer_size __save
    #pragma __required_pointer_size __long

    /* Define a type for a 64-bit pointer to a SETMODE_BUF structure.
     */
    typedef SETMODE_BUF *SETMODE_BUF_PQ;

    #pragma __required_pointer_size __restore

    /* This must be a pointer to a 64-bit address since it will be containing
     *  the address of a user buffer which may be a 64-bit or a 32-bit value.
     */
    SETMODE_BUF_PQ setmode_bufp;

    /* The caller passes the address of their setmode buffer in the $QIO P1
     * parameter.
     */
    setmode_bufp = (SETMODE_BUF_PQ) irp->irp$q_qio_p1;

    /* Assure that the caller's setmode buffer is readable by the caller.
     * If not, abort the I/O request now with an ACCVIO status and return
     * back to the FDT dispatcher in the $QIO system service.
     */
    if (! ( __PAL_PROBER (setmode_bufp, sizeof(SETMODE_BUF)-1, irp->irp$b_rmod) ))
        return ( call_abortio (irp, pcb, (UCB *)ucb, SS$_ACCVIO) );

    /* If function is SETCHAR then set dev class and type */

    if (irp->irp$v_fcode == IO$_SETCHAR) {
        ucb->ucb$r_ucb.ucb$b_devclass = setmode_bufp->devclass;
        ucb->ucb$r_ucb.ucb$b_devtype = setmode_bufp->devtype;
    }

    /* Set the default buffer and device dependent characteristics */
```

```
    ucb->ucb$r_ucb.ucb$w_devbufsiz = setmode_bufp->devbufsiz;
    ucb->ucb$r_ucb.ucb$l_devdepend = setmode_bufp->devdepend;

    /* Finish the IO; return SS$_FDT_COMPL to the FDT dispatcher in the $QIO
     * system service.
     */
    return ( call_finishio (irp, (UCB *)ucb, SS$_NORMAL, 0) );
}


/*
 * LR$WRITE - FDT Routine for Write Function Codes
 *
 * Functional description:
 *
 *   This routine is called by the FDT dispatcher in the $QIO system service
 *   to process write functions.  This FDT routine validates the request,
 *   allocates a buffered I/O packet, formats and copies the contents of the
 *   user buffer into the buffered I/O packet, and queues the IRP to this
 *   driver's start I/O routine.  The user buffer address is contained in
 *   irp$q_qio_p1 ($QIO P1 parameter) on input and will be treated as a
 *   64-bit address.
 *
 *   When the IRP is successfully queued to the driver's start I/O routine,
 *   irp$ps_bufio_pkt points to the buffered I/O packet, irp$l_boff is the
 *   number of bytes that have been charged against the process, and irp$l_bcnt
 *   is the actual count of data bytes in the buffered I/O packet that are
 *   to be sent to the printer.  Note that the contents of the irp$ps_bufio_pkt
 *   and irp$l_boff cells must not be changed since I/O post processing will
 *   use these to deallocate the buffer packet and to credit the process.
 *
 *   Since this is an upper-level FDT routine, this routine always returns
 *   the SS$_FDT_COMPL status.  The $QIO status that is to be returned to
 *   the caller of the $QIO system service is returned indirectly by the
 *   FDT completion routines (e. g. exe_std$abortio, exe_std$qiodrvpkt) via
 *   the FDT context structure.
 *
 * Calling convention:
 *
 *   status = lr$write (irp, pcb, ucb, ccb)
 *
 * Input parameters:
 *
 *   irp        Pointer to I/O request packet
 *   pcb        Pointer process control block
 *   ucb        Pointer to unit control block
 *   ccb        Pointer to channel control block
 *
 * Output parameters:
 *
 *   None.
 *
```

```
 * Return value:
 *
 *   status     SS$_FDT_COMPL
 *
 * Environment:
 *
 *   Kernel mode, user process context, IPL 2.
 */

int lr$write (IRP *irp, PCB *pcb, LR_UCB *ucb, CCB *ccb) {

    CHAR_PQ qio_bufp;           /* 64-bit pointer to caller's buffer */
    int qio_buflen;             /* Number of bytes in caller's buffer */
    BUFIO *sys_bufp;            /* Pointer to a system buffer packet */
    int32 sys_buflen;           /* Computed required system packet size */
    int   sys_bufspace;         /* Actual space in system buffer for data */
    char *sys_datap;            /* Working pointer to next byte in sysbuf */
    int pass_all;               /* True if this is a "pass all" write */
    int carcon_count;
    char carcon_char;
    int status;
    int tmp_status;

    FMT_DATA fmt_data;          /* Formatting status information */

    /* Get the pointer to the caller's buffer and the size of the caller's
     * buffer from the $QIO P1 and P2 parameters respectively.  The caller's
     * buffer is treated as a 64-bit address although it may be a 32-bit
     * address.
     */
    qio_bufp   = (CHAR_PQ)irp->irp$q_qio_p1;
    qio_buflen = irp->irp$l_qio_p2;

    /* Assure that the caller has read access to this buffer to do a write
     * operation.  If abort the I/O request and return the SS$_FDT_COMPL
     * warning status.  If this is the case, we must return back to the FDT
     * dispatcher in the $QIO system service.  Note we continue on even if
     * the user buffer is zero length since there may be * carriage control
     * to output.
     */
    if (qio_buflen != 0) {
        status = exe_std$writechk (irp, pcb, &(ucb->ucb$r_ucb),
                                   qio_bufp, qio_buflen);
        if ( ! $VMS_STATUS_SUCCESS(status) )
           return (call_abortio (irp, pcb, (UCB *)ucb, status));
    }

    /* Start out assuming that the required system buffer packet size is
     * the size of the $QIO buffer plus the size of the 64-bit buffer
     * packet header.
     */
    sys_buflen = qio_buflen + BUFIO$K_HDRLEN64;
```

```
/* This is a "pass all" request either if the write physical function
 * was specified or if the device is set to "write pass all" mode.
 */
pass_all = irp->irp$v_func == IO$_WRITEPBLK  ||
           (ucb->ucb$r_ucb.ucb$l_devdepend & LP$M_PASSALL);

/* If this is not a "pass all" request, then interpret the $QIO P4
 * carriage control parameter.  Adjust the required system buffer packet
 * size by the prefix and suffix counts plus room of data expansion.
 * Currently, the only expansion possible is an extra CR in the prefix
 * and suffix characters if "new line" was specified.
 */
if (pass_all)
{
   /* Allocate a system buffer for the data in the user buffer.  If this
    * fails then abort the I/O request and return back to the FDT dispatcher
    * in the $QIO system service.  Otherwise, exe_std$alloc_bufio_64 will
    * point irp$ps_bufio_pkt (overlays irp$l_svapte) to the bufio packet
    * and irp$l_boff to the number of bytes charged.
    */

   status = exe_std$alloc_bufio_64(irp,
                                   pcb,
                                   (VOID_PQ) qio_bufp,  /* user buffer    */
                                   sys_buflen);  /* buff size plus header */

   if ( ! $VMS_STATUS_SUCCESS(status) )
       return ( call_abortio (irp, pcb, (UCB *)ucb, status) );

   /* sys_bufp points to the bufio header packet.                          */
   sys_bufp         = irp->irp$ps_bufio_pkt;

   /* sys_datap points to the first free data byte in the buffer packet. */
   sys_datap        = sys_bufp->bufio$ps_pktdata;

   /* Copy the contents of the user buffer to the bufio data area.        */
   memcpy (sys_datap, qio_bufp, qio_buflen);

   irp->irp$l_bcnt  = qio_buflen;
}
else
{
    /* These next steps only need to be done once before formatting the
     * buffer.
     */
    irp->irp$l_iost2 = irp->irp$l_qio_p4;
    exe_std$carriage (irp);
    sys_buflen += ((CARCON *) &irp->irp$b_carcon)->prefix_count +
                  ((CARCON *) &irp->irp$b_carcon)->suffix_count;
```

```
/* When we format the buffer it is possible that the buffer we allocate
 * will not be large enough.  So we allocate a system buffer and try to
 * format users buffer into it.  If it does not fit we will deallocate
 * the buffer and return the quota and try a larger buffer.  If it fits
 * we will update the row and column data and drop out of the format
 * loop.
 */

fmt_data.page_length = (int) ucb->ucb$r_ucb.ucb$b_vertsz;
fmt_data.page_width =  (int) ucb->ucb$r_ucb.ucb$w_devbufsiz;

do
{
   sys_buflen += DATA_EXPND_CUSHION;

   status = exe_std$alloc_bufio_64(irp,
                                   pcb,
                                   (VOID_PQ) qio_bufp, /* user buffer */
                                   sys_buflen);/* buf siz plus header */

   if ( ! $VMS_STATUS_SUCCESS(status) )
       return ( call_abortio (irp, pcb, (UCB *)ucb, status) );

   /* sys_bufp points to the bufio header packet.                    */
   sys_bufp = irp->irp$ps_bufio_pkt;

   /* sys_buflen is the number of bytes charged by alloc_bufio.      */
   sys_buflen = sys_bufp->bufio$w_size;

   fmt_data.cr_pend = ucb->ucb$l_lr_cr_pend;
   fmt_data.sys_datap = (char *) sys_bufp->bufio$ps_pktdata;
   fmt_data.buffer_space = sys_buflen - BUFIO$K_HDRLEN64;
   fmt_data.column_pos = ucb->ucb$l_lr_cursor;
   fmt_data.line_on_page = ucb->ucb$l_lr_lincnt;
   fmt_data.total_lines = 0;
   fmt_data.total_bytes = 0;

   /* Expand the prefix carriage control into the allocated system
    * buffer.  If the carriage control count is non-zero and the
    * carriage control character is 0, this means "new line."  Output
    * an initial CR, then the counted number of LFs.
    */
```

```
carcon_count = ((CARCON *) &irp->irp$b_carcon)->prefix_count;
if (carcon_count != 0) {
   carcon_char = ((CARCON *) &irp->irp$b_carcon)->prefix_char;
   if (carcon_char == 0) {
      status = lr$format_char (ucb, CR, &fmt_data);
      carcon_char = LF;
   }
   while ((status & SS$_NORMAL) && (carcon_count > 0))
   {
      status = lr$format_char(ucb, carcon_char, &fmt_data);
      carcon_count -= 1;
   }
}

/* If no error so far then format the users buffer */

carcon_count = 0;
while ((status & SS$_NORMAL) && (carcon_count < qio_buflen))
{
   status = lr$format_char(ucb, qio_bufp[carcon_count], &fmt_data);
   carcon_count += 1;
}

/* Expand the suffix carriage control into the allocated system
 * buffer.  If the carriage control count is non-zero and the
 * carriage control character is 0, this means "new line."  Output
 * an initial CR, then the counted number of LFs.
 */
carcon_count = ((CARCON *) &irp->irp$b_carcon)->suffix_count;
if ((carcon_count != 0) && (status & SS$_NORMAL)) {
   carcon_char = ((CARCON *) &irp->irp$b_carcon)->suffix_char;
   if (carcon_char == 0) {
      status = lr$format_char(ucb, CR, &fmt_data);
      carcon_char = LF;
   }
   while ((status & SS$_NORMAL) && (carcon_count > 0))
   {
      status = lr$format_char(ucb, carcon_char, &fmt_data);
      carcon_count -= 1;
   }
}

/* If an error has occured then we need to delete the buffer so
 * we can try to get a larger buffer and try to format it once
 * again.
 */

if (!($VMS_STATUS_SUCCESS(status)))
{
   exe_std$credit_bytcnt(irp->irp$l_boff, pcb);
   irp->irp$ps_bufio_pkt = (void *) 0;
   irp->irp$l_boff = 0;
   tmp_status = exe_std$deanonpaged((void *)sys_bufp);
}
```

```
        } while (! $VMS_STATUS_SUCCESS(status));

        ucb->ucb$l_lr_cr_pend = fmt_data.cr_pend;
        ucb->ucb$l_lr_cursor = fmt_data.column_pos;
        ucb->ucb$l_lr_lincnt = fmt_data.line_on_page;
        irp->irp$l_iost2 = fmt_data.total_lines;
        irp->irp$l_bcnt = fmt_data.total_bytes;
    }

    /* If characters to be output Queue this I/O request to the start I/O
     * routine and return SS$_FDT_COMPL back to the FDT dispatcher in the
     * $QIO system service.  If not then just finish the request, it is
     * possible that there will be no output if the printer is set to truncate
     * and we are already at the right margin when a new output is started.
     */
    if (irp->irp$l_bcnt)
    {
       return ( call_qiodrvpkt (irp, (UCB *)ucb) );
    }
    else
    {
       return ( call_finishio (irp, (UCB *)ucb, SS$_NORMAL, 0) );
    }
}


/*
 * LR$FORMAT_CHAR - This routine is used to format users data
 *
 * Functional description:
 *
 *
 *      This routine determines if any special action needs to be taken based
 * on what the character is and how the printer port is configured.
 * Additionally, it handles truncating output or wrapping output, as well
 * as tabs, line feeds, form feeds, and carriage return.
 *
 * Calling convention:
 *
 *   status = lr$format_char (ucb, out_char, fmt_data)
 *
 * Input parameters:
 *
 *   ucb            Pointer to UCB for this device
 *   out_char       Character to be output
 *   fmt_data       Data structure with a variety of data
 *
 * Output parameters:
 *
 *   none
 *
 * Return value:
```

```
 *
 *   status  SS$NORMAL       - Buffer filled with no problem
 *           SS$_TOOMUCHDATA - Buffer to small could not format all the data
 *
 * Environment:
 *
 *   Kernel mode, user process context, IPL 2.
 */

int  lr$format_char (LR_UCB *ucb, unsigned char out_char, FMT_DATA *fmt_data)
{

unsigned char    tmp_char;

int     char_mask;          /* Bit in array segment for this character */
int     fill_chars;         /* Number filler chracters needed */
int     i;                  /* temporary counter */
int     index;              /* Index into array of character characteristics */
int     status = SS$_NORMAL;
int     tab_stop;           /* Next tab stop position */

if (fmt_data->cr_pend)
{
   fmt_data->cr_pend = FALSE;
   if ((out_char !=  FF) && (out_char != VT) && (out_char < DEL))
   {
      tmp_char = out_char;
      fmt_data->column_pos = 0;
      if (fmt_data->total_bytes++ < fmt_data->buffer_space)
      {
         *fmt_data->sys_datap++ = CR;
         status = lr$format_char(ucb, tmp_char, fmt_data);
         return(status);
      }
      else
         return(SS$_TOOMUCHDATA);
   }
}

/* Compute character array index and bit position.  This is done to make it
   easy to see if character is considerd a control character or something
   that can be upcased */

   index = (int) out_char/32;
   char_mask = 1 << (int) out_char%32;
```

```
if (CTRL_TABLE[index] & char_mask)
{
   if ((out_char >= DEL) && (!(ucb->ucb$r_ucb.ucb$l_devdepend & LP$M_PRINTALL)))
   {
       return (SS$_NORMAL);                        /* Drop character */
   }
   else if (out_char == CR)                        /* CR */
   {
      if (!(ucb->ucb$r_ucb.ucb$l_devdepend & LP$M_CR))
      {
         fmt_data->cr_pend = TRUE;
         return (SS$_NORMAL);
      }
      else
      {
         if (fmt_data->total_bytes++ < fmt_data->buffer_space)
         {
            fmt_data->column_pos = 0;
            *fmt_data->sys_datap++ = CR;
            return(SS$_NORMAL);
         }
         else
            return(SS$_TOOMUCHDATA);
      }
   }
   else if (out_char == HT)                        /* TAB */
   {
      if (!(ucb->ucb$r_ucb.ucb$l_devdepend & LP$M_TAB))
      {
         tab_stop = (fmt_data->column_pos + 8) & ~7;
         fill_chars = tab_stop - fmt_data->column_pos;
         i = 0;
         while ((status & SS$_NORMAL) && (i < fill_chars))
         {
            status = lr$format_char(ucb, SP, fmt_data);
            i += 1;
         }
         return (status);
      }
   }
   else if (out_char == VT)                        /* VT */
   {
      if (!(ucb->ucb$r_ucb.ucb$l_devdepend & LP$M_PRINTALL))
      {
         return (SS$_NORMAL);                      /* Drop character */
      }
   }
   else if (out_char == FF)                        /* FF */
   {
      fill_chars = fmt_data->page_length - fmt_data->line_on_page;
      if (ucb->ucb$r_ucb.ucb$l_devdepend & LP$M_MECHFORM)
      {
```

```
            fmt_data->total_lines = fmt_data->total_lines + fill_chars;
            fmt_data->line_on_page = 0;
         }
         else
         {
            i = 0;
            while ((status & SS$_NORMAL) && (i < fill_chars))
            {
               status = lr$format_char(ucb, LF, fmt_data);
               i += 1;
            }
            return (status);
         }
      }
      else if (out_char == LF)                        /* LF */
      {
         if (fmt_data->total_bytes++ < fmt_data->buffer_space)
         {
            *fmt_data->sys_datap++ = LF;
            fmt_data->line_on_page += 1;
            fmt_data->total_lines += 1;
            fmt_data->column_pos = 0;
            if (fmt_data->line_on_page >= fmt_data->page_length)
            {
               fmt_data->line_on_page = 0;
            }
            return(SS$_NORMAL);
         }
         else
            return(SS$_TOOMUCHDATA);
      }
      else                                            /* Other control chars */
      {
         if (!(ucb->ucb$r_ucb.ucb$l_devdepend & LP$M_PRINTALL))
         {
            return (SS$_NORMAL);                      /* Drop character */
         }
      }
   }
   else if (!(ucb->ucb$r_ucb.ucb$l_devdepend & LP$M_LOWER))
   {
      if (CASE_TABLE[index] & char_mask)              /* Character is lower case */
      {
         out_char = out_char - SP;
      }
   }

   /* If here we have a character to output see if room to do so. If space and if
      FALLBACK is set then translate it */
```

```
if (fmt_data->column_pos > fmt_data->page_width)
{
   if ((ucb->ucb$r_ucb.ucb$l_devdepend & LP$M_TRUNCATE) &&
       (!(ucb->ucb$r_ucb.ucb$l_devdepend & LP$M_WRAP)))
   {
       return (SS$_NORMAL);
   }
   else
   {
      status= lr$format_char(ucb, CR, fmt_data);
      if (status & SS$_NORMAL)
      {
         status= lr$format_char(ucb, LF, fmt_data);
         if (!(status & SS$_NORMAL)) return (status);
      }
   }
}
fmt_data->column_pos +=1;
if (fmt_data->total_bytes++ < fmt_data->buffer_space)
{
   if (!(ucb->ucb$r_ucb.ucb$l_devdepend & LP$M_FALLBACK))
   {
      *fmt_data->sys_datap++ = out_char;
   }
   else
   {
      *fmt_data->sys_datap++ = TRANS_TABLE[out_char];
   }
   return (SS$_NORMAL);
}
else
{
   return (SS$_TOOMUCHDATA);
}

}
```

```
/*
 * LR$STARTIO - Start I/O Routine
 *
 * Functional description:
 *
 *   This routine is the driver start I/O routine.  This routine is called
 *   by ioc_std$initiate to process the next I/O request that has been
 *   queued to this device.  For this driver, the only function that is
 *   passed to the start I/O routine is a write operation.
 *
 *   Before this routine is called, ucb$v_cancel, ucb$v_int, ucb$v_tim, and
 *   ucb$v_timout are cleared. The ucb$l_svapte, ucb$l_boff, and ucb$l_bcnt
 *   cells are set in  ioc_std$initiate from their corresponding IRP cells.
 *   Unlike their IRP counterparts, these UCB cells are working storage and
 *   can be changed by a driver.  This driver uses ucb$l_svapte to point to
 *   the next byte to output in the system buffer packet, and irp$l_bcnt to
 *   keep the count of the remaining bytes to output.
 *
 *   This routine acquires the device lock and raises IPL to device IPL.
 *   The device lock is restored and the original IPL is restored via wfikpch
 *   before this routine returns to its caller.
 *
 * Calling convention:
 *
 *   lr$startio (irp, ucb)
 *
 * Input parameters:
 *
 *   irp        Pointer to I/O request packet
 *   ucb        Pointer to unit control block
 *
 * Output parameters:
 *
 *   None.
 *
 * Return value:
 *
 *   None.
 *
 * Environment:
 *
 *   Kernel mode, system context, fork IPL, fork lock held.
 */
void lr$startio (IRP *irp, LR_UCB *ucb) {

    int orig_ipl;

    /* Adjust ucb$l_svapte such that it points to the start of the data in
     * the system buffer packet.
     */
    ucb->ucb$r_ucb.ucb$l_svapte = (char *) ucb->ucb$r_ucb.ucb$l_svapte +
                                  BUFIO$K_HDRLEN64;
```

```
    /* Acquire the device lock, raise IPL, saving original IPL */

    device_lock (ucb->ucb$r_ucb.ucb$l_dlck, RAISE_IPL, &orig_ipl);

    /* Send the first character to the device.  We can ignore the status,
     * since we will timeout if the device is not ready.
     */
    lr$send_char_dev (ucb);

    /* Set up a wait for the completion of the I/O by using the wfikpch macro.
     * Wfikpch will restore the device lock and restore IPL.  When output of
     * the entire buffer has been completed, the lr$interrupt routine will
     * queue the lr$iodone_fork routine.  If the I/O does not complete within
     * LR_WFI_TMO seconds, then exe$timeout will call lr$wfi_timeout.
     */
    wfikpch (lr$iodone_fork, lr$wfi_timeout, irp, 0, ucb, LR_WFI_TMO, orig_ipl);

    return;
}


/*
 * LR$SEND_CHAR_DEV - Send Character to the Device
 *
 * Functional description:
 *
 *   This routine sends the next character from the system buffer to the
 *   device via the printer write data register.  This routine decrements the
 *   count of remaining bytes (ucb$l_bcnt) and advances the pointer to the
 *   next character (ucb$l_svapte). (ucb$l_svapte was made to point to the
 *   bufio data packet area in LR$STARTIO by adding the header length to the
 *   original bufio header pointer.)
 *
 *   This is an internal routine that is used by the start I/O, interrupt
 *   service, and periodic check device ready routines.
 *
 * Calling convention:
 *
 *   status = lr$send_char_dev (ucb)
 *
 * Input parameters:
 *
 *   ucb        Pointer to unit control block
 *
 * Output parameters:
 *
 *   None.
 *
 * Return value:
 *
 *   status     SS$_NORMAL      if the next data byte was sent to the printer
 *                              device.
 *              SS$_DEVOFFLINE  if the next data byte was not sent to the
```

```
*                               printer device since it is not ready to accept
*                               data.
*
* Environment:
*
*   Kernel mode, system context, device IPL, device lock held.
*/

static int lr$send_char_dev (LR_UCB *ucb) {

    int device_data;            /* Data from or for CRAM */
    char *sys_datap;            /* Pointer to next byte in buffer packet */

    /* Set the Port Control Register.
     * Set the INIT_OFF bit to disable the "INIT" signal.  Set the IRQ_EN bit
     * to enable interrupts.  Assure that the STROBE bit is clear so that we
     * can cause a 0-to-1 transition after loading the data register.  Assure
     * that the DIR_READ bit is clear since we are doing writes to the data
     * register.  Note byte-lane shift if ISA option.
     */
    if (ucb->ucb$l_lr_jensen)
        device_data = LPC_M_INIT_OFF | LPC_M_IRQ_EN;
    else
        device_data = (LPC_M_INIT_OFF | LPC_M_IRQ_EN) << 16;

    ucb->ucb$ps_cram_lcw->cram$q_wdata = device_data;
    ioc$cram_io (ucb->ucb$ps_cram_lcw);

    /* Read the port status register.  Note byte-lane shift if ISA option. */

    ioc$cram_io (ucb->ucb$ps_cram_lps);
    device_data = ucb->ucb$ps_cram_lps->cram$q_rdata;
    if ( ! ucb->ucb$l_lr_jensen) device_data >>= 8;

    /* If the device is not ready to accept a character, then do not attempt
     * to send it.  Return an error status.
     */
    if (   ((LPS *) &device_data)->lps_paperout ||    /* paper out */
         ! ((LPS *) &device_data)->lps_ok       ||    /* not ok, i.e. error */
         ! ((LPS *) &device_data)->lps_online   ||    /* not online */
         ! ((LPS *) &device_data)->lps_ready    )     /* not ready */
        return SS$_DEVOFFLINE;

    /* The device is ready.  Load the data byte.  Update ucb$l_svapte to
     * point to the next byte and decrement the count of bytes lef in
     * ucb$l_bcnt.  Note that no byte-lane shift is necessary for this register.
     */
    sys_datap = (char *) ucb->ucb$r_ucb.ucb$l_svapte;
    device_data = *sys_datap++;
    ucb->ucb$r_ucb.ucb$l_svapte = (void *) sys_datap;
    ucb->ucb$r_ucb.ucb$l_bcnt--;
    ucb->ucb$ps_cram_lwd->cram$q_wdata = device_data;
    ioc$cram_io (ucb->ucb$ps_cram_lwd);
```

```
    /* Latch the data byte to the printer.
     * Because some printers trigger on the 0 to 1 transistion of STROBE and
     * other trigger on the 1 to 0 transitions we have to write to the line
     * control register twice.  INIT_OFF and IRQ_EN were set earlier and are
     * kept set. DIR_READ is kept clear.  Note byte-lane shift if ISA option.
     */
    if (ucb->ucb$l_lr_jensen)
        device_data = LPC_M_INIT_OFF | LPC_M_IRQ_EN | LPC_M_STROBE;
    else
        device_data = (LPC_M_INIT_OFF | LPC_M_IRQ_EN | LPC_M_STROBE) << 16;

    ucb->ucb$ps_cram_lcw->cram$q_wdata = device_data;
    ioc$cram_io (ucb->ucb$ps_cram_lcw);

    if (ucb->ucb$l_lr_jensen)
        device_data = LPC_M_INIT_OFF | LPC_M_IRQ_EN;
    else
        device_data = (LPC_M_INIT_OFF | LPC_M_IRQ_EN) << 16;

    ucb->ucb$ps_cram_lcw->cram$q_wdata = device_data;
    ioc$cram_io (ucb->ucb$ps_cram_lcw);

    /* Data byte sent.  Return success. */

    return SS$_NORMAL;
}


/*
 * LR$INTERRUPT - Interrupt Service Routine
 *
 * Functional description:
 *
 *   This is the interrupt service routine for the parallel line printer
 *   port.  This routine is called by the system interrupt dispatcher.
 *
 *   This routine will attempt to send the next character to the device
 *   until either there are no more characters left or the I/O is canceled.
 *   At which point, this routine will queue the lr$iodone_fork routine
 *   which was set up either in lr$startio or lr$check_ready_fork.
 *
 *   If the interrupt is not expected by an active I/O on this device then
 *   it is simply dismissed.
 *
 *
 * Calling convention:
 *
 *   lr$interrupt (idb)
 *
 * Input parameters:
 *
 *   idb        Pointer to interrupt dispatch block
 *
 * Output parameters:
```

```
 *
 *   None.
 *
 * Return value:
 *
 *   None.
 *
 * Environment:
 *
 *   Kernel mode, system context, device IPL.
 *
 */

void lr$interrupt (IDB *idb)  {

    LR_UCB *ucb;
    int device_data;            /* Data from or for CRAM */
    int status;

    /* Get the UCB from the IDB owner field which was set up by the lr$unit_init
     * routine.
     */
    ucb = (LR_UCB *) idb->idb$ps_owner;

    /* Acquire the device lock.  We are already at device IPL */

    device_lock (ucb->ucb$r_ucb.ucb$l_dlck, NORAISE_IPL, NOSAVE_IPL);

    /* If interrupt is expected, then process it, otherwise ignore it */

    if (ucb->ucb$r_ucb.ucb$v_int) {

        /* If there are characters left and the I/O has not been cancelled
         * then attempt to send the next character.  There is no need to check
         * the status since the interrupt timeout will expire if the device is
         * not ready.  Otherwise, queue the I/O done fork routine that was
         * setup via wfikpch.
         */
        if (ucb->ucb$r_ucb.ucb$l_bcnt > 0  &&  ! ucb->ucb$r_ucb.ucb$v_cancel) {
            lr$send_char_dev (ucb);
        } else {
            ucb->ucb$r_ucb.ucb$v_int = 0;
            ucb->ucb$r_ucb.ucb$v_tim = 0;
            exe_std$queue_fork( (FKB *)ucb );
        }
    }

    /* Restore the device lock, stay at device IPL */

    device_unlock (ucb->ucb$r_ucb.ucb$l_dlck, NOLOWER_IPL, SMP_RESTORE);

    /* return back to interrupt dispatcher */

    return;
}
```

```
/*
 * LR$IODONE_FORK - I/O Completion Fork Routine
 *
 * Functional description:
 *
 *   This is the fork routine which passes the current I/O request on to
 *   I/O postprocessing.  This routine is queued by the interrupt service
 *   routine when the I/O request has been completed.  This routine can also
 *   be called directly from lr$check_ready_fork if the I/O request is
 *   cancelled while it is stalled due to an offline condition.
 *
 * Calling convention:
 *
 *   lr$iodone_fork (irp, not_used, ucb)
 *
 * Input parameters:
 *
 *   irp        Pointer to I/O request packet
 *   not_used   Unused fork routine parameter fr4
 *   ucb        Pointer to unit control block
 *
 * Output parameters:
 *
 *   None.
 *
 * Return value:
 *
 *   None.
 *
 * Environment:
 *
 *   Kernel mode, system context, fork IPL, fork lock held.
 */
void lr$iodone_fork (IRP *irp, void *not_used, LR_UCB *ucb)
{
    int status = SS$_NORMAL;            /* Assume everything went ok *

    /* If the request was cancelled or timed out of its own accord then
     * set the status accordingly.
     */
    if (ucb->ucb$r_ucb.ucb$v_cancel) {
        status = SS$_ABORT;
    } else if (ucb->ucb$r_ucb.ucb$v_timout) {
        status = SS$_TIMEOUT;
    }

    /* Send this I/O request to I/O post processing *

    ioc_std$reqcom (status, 0, &(ucb->ucb$r_ucb));
    return;
}
```

```
/*
 * LR$WFI_TIMEOUT - Wait-for-interrupt timeout routine
 *
 * Functional description:
 *
 *   This routine is the wait-for-interrupt timeout routine.  It is called
 *   by exe$timeout when an operation set up by wfikpch takes more that the
 *   specified number of seconds.
 *
 *   This routine queues a fork routine, lr$check_ready_fork, to handle
 *   periodic checking of the readiness of the device to resume output and
 *   to issue periodic "device offline" messages via OPCOM.
 *
 * Calling convention:
 *
 *   lr$wfi_timeout (irp, not_used, ucb)
 *
 * Input parameters:
 *
 *   irp        Pointer to I/O request packet
 *   not_used   Unused fork routine parameter fr4
 *   ucb        Pointer to unit control block
 *
 * Output parameters:
 *
 *   None.
 *
 * Return value:
 *
 *   None.
 *
 * Environment:
 *
 *   Kernel mode, system context, device IPL, fork lock held, device lock held.
 *
 */
void lr$wfi_timeout (IRP *irp, void *not_used, LR_UCB *ucb) {

    /* A wait-for-interrupt has timed out.  Count the device as having been
     * offline for the duration of the wait-for-interrupt interval.
     */
    ucb->ucb$l_lr_oflcnt = LR_WFI_TMO;

    /* Queue a fork-wait thread that checks once a second for the device being
     * ready to accept data.  One reason for deferring this work to fork level
     * is that exe_std$sndevmsg cannot be called at device IPL.
     */
    fork_wait (lr$check_ready_fork, irp, 0, ucb);

    return;
}
```

```
/*
 * LR$CHECK_READY_FORK - Periodic Check for Device Ready
 *
 * Functional description:
 *
 *   This routine performs a once-a-second check of the readiness of the
 *   device to resume output.  While the device remains offline this fork
 *   routine reschedules itself via the fork wait queue.  When the device
 *   is ready to resume, the next character is sent and the remainder of
 *   the output is done by the interrupt service routine.
 *
 *   If the device remains offline for ucb$l_lr_msg_tmo seconds (initially
 *   set to LR_OFFLINE_TMO) then a "device offline" message is sent to
 *   OPCOM.  The device offline message interval is doubled each time while
 *   it is less than an hour.  When the device becomes ready again, the offline
 *   message interval is reset to its initial LR_OFFLINE_TMO value.
 *
 * Calling convention:
 *
 *   lr$check_ready_fork  (irp, not_used, ucb)
 *
 * Input parameters:
 *
 *   irp        Pointer to I/O request packet
 *   not_used   Unused fork routine parameter fr4
 *   ucb        Pointer to unit control block
 *
 * Output parameters:
 *
 *   None.
 *
 * Return value:
 *
 *   None.
 *
 * Environment:
 *
 *   Kernel mode, system context, fork IPL, fork lock held.
 */
void lr$check_ready_fork (IRP *irp, void *not_used, LR_UCB *ucb) {
    int orig_ipl;
    int status;
```

```
/* If the I/O request has been canceled while we've been waiting or there
 * are no more characters to send to the device then call our I/O done fork
 * routine directly to complete the I/O request and then return from this
 * routine.
 */
if (ucb->ucb$r_ucb.ucb$v_cancel || ucb->ucb$r_ucb.ucb$l_bcnt == 0) {
    lr$iodone_fork (irp, 0, ucb);
    return;
}

/* Acquire the device lock, raise IPL, saving original IPL */

device_lock (ucb->ucb$r_ucb.ucb$l_dlck, RAISE_IPL, &orig_ipl);

/* Attempt to send the next character to the device.  If the device is
 * still not ready, then the character will not be sent and an error status
 * will be returned.
 */
status = lr$send_char_dev (ucb);

/* If we successfully sent a character to the device then we're back in
 * business.  Set up a wait for the completion of the I/O via wfikpch
 * just like our start I/O routine.  Wfikpch will restore the device lock
 * and restore IPL.  But first, clear the offline count and set the offline
 * message interval to its initial value.  And, return from this routine.
 */
if ( $VMS_STATUS_SUCCESS(status) ) {
    ucb->ucb$l_lr_msg_tmo = LR_OFFLINE_TMO;
    ucb->ucb$l_lr_oflcnt = 0;
    wfikpch (lr$iodone_fork, lr$wfi_timeout, irp, 0, ucb,
             LR_WFI_TMO, orig_ipl);
    return;
}

/* Otherwise, the device is still offline.  Increment the offline time. */

ucb->ucb$l_lr_oflcnt++;

/* Restore the device lock, return to the original entry IPL */

device_unlock (ucb->ucb$r_ucb.ucb$l_dlck, orig_ipl, SMP_RESTORE);

/* If the offline count has reached the "device offline" message interval
 * then it's time to send it to OPCOM and start a new offline interval.
 * If this message interval was less than an hour, double the next one.
 */
if (ucb->ucb$l_lr_oflcnt >= ucb->ucb$l_lr_msg_tmo) {

    extern MB_UCB *sys$ar_oprmbx;           /* Pointer to OPCOM mbx ucb */

    exe_std$sndevmsg (sys$ar_oprmbx, MSG$_DEVOFFLIN, &(ucb->ucb$r_ucb));

    ucb->ucb$l_lr_oflcnt = 0;
    if (ucb->ucb$l_lr_msg_tmo < ONE_HOUR)
        ucb->ucb$l_lr_msg_tmo *= 2;
}
```

```
    /* Setup to check the device again in one second via the fork-wait queue */
    fork_wait (lr$check_ready_fork, irp, 0, ucb);
    return;
}
```

B.2 LRDRIVER.H

This section contains the LRDRIVER.H file, which is the fallback table remapping for DEC Multinational Character Set "MCS".

```
/*  X-1
**
**   Copyright © Digital Equipment Corporation, 1994  All Rights Reserved.
**   Unpublished rights reserved under the copyright laws of the United States.
**
**   The software contained on this media is proprietary to and embodies the
**   confidential technology of Digital Equipment Corporation.  Possession, use,
**   duplication or dissemination of the software and media is authorized only
**   pursuant to a valid written license from Digital Equipment Corporation.
**
**   RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure by the U.S.
**   Government is subject to restrictions as set forth in Subparagraph
**   (c)(1)(ii) of DFARS 252.227-7013, or in FAR 52.227-19, as applicable.
**
**
**++
**  FACILITY:
**
**      Example Device Driver for OpenVMS AXP
**
**  MODULE DESCRIPTION:
**
**      This module contains the fallback table remapping for DEC Multinational
**      Character Set "MCS".
**
**  AUTHOR:
**
**      OpenVMS Alpha Development Group
**
**  MODIFICATION HISTORY:
**
**      X-1     VMS000          OpenVMS Alpha Drivers   20-July-1994
**              Initial version.
**
**--
*/
```

```
/*
*+
*       Build up a table with the whole ASCII 8-Bit character set.  This table
* is used to translate 8-Bit characters to 7-Bit characters if the printer is
* doing character fallback translation.  This only exists for to keep the
* printer code equivalent to that in the old LP11 interface.  Any modern
* printer that is likely to be controlled by LRDRIVER should be capable of
* deaing with 8-Bit characters.
*-
*/
static unsigned char    TRANS_TABLE[256] =  {
'\x00',         /* NUL   */
'\x01',         /* SOH   */
'\x02',         /* STX   */
'\x03',         /* ETX   */
'\x04',         /* EOT   */
'\x05',         /* ENQ   */
'\x06',         /* ACK   */
'\x07',         /* BEL   */
'\x08',         /* BS    */
'\x09',         /* HT    */
'\x0A',         /* LF    */
'\x0B',         /* VT    */
'\x0C',         /* FF    */
'\x0D',         /* CR    */
'\x0E',         /* SO    */
'\x0F',         /* SI    */
'\x10',         /* DLE   */
'\x11',         /* DC1   */
'\x12',         /* DC2   */
'\x13',         /* DC3   */
'\x14',         /* DC4   */
'\x15',         /* NAK   */
'\x16',         /* SYN   */
'\x17',         /* ETB   */
'\x18',         /* CAN   */
'\x19',         /* EM    */
'\x1A',         /* SUB   */
'\x1B',         /* ESC   */
'\x1C',         /* FS    */
'\x1D',         /* GS    */
'\x1E',         /* RS    */
'\x1F',         /* US    */
'\x20',         /* SP    */
'\x21',         /* !     */
'\x22',         /* "     */
'\x23',         /* #     */
'\x24',         /* $     */
'\x25',         /* %     */
'\x26',         /* &     */
'\x27',         /* '     */
'\x28',         /* (     */
```

```
'\x29',         /* )     */
'\x2A',         /* *     */
'\x2B',         /* +     */
'\x2C',         /* ,     */
'\x2D',         /* -     */
'\x2E',         /* .     */
'\x2F',         /* /     */
'\x30',         /* 0     */
'\x31',         /* 1     */
'\x32',         /* 2     */
'\x33',         /* 3     */
'\x34',         /* 4     */
'\x35',         /* 5     */
'\x36',         /* 6     */
'\x37',         /* 7     */
'\x38',         /* 8     */
'\x39',         /* 9     */
'\x3A',         /* :     */
'\x3B',         /* ;     */
'\x3C',         /* <     */
'\x3D',         /* =     */
'\x3E',         /* >     */
'\x3F',         /* ?     */
'\x40',         /* @     */
'\x41',         /* A     */
'\x42',         /* B     */
'\x43',         /* C     */
'\x44',         /* D     */
'\x45',         /* E     */
'\x46',         /* F     */
'\x47',         /* G     */
'\x48',         /* H     */
'\x49',         /* I     */
'\x4A',         /* J     */
'\x4B',         /* K     */
'\x4C',         /* L     */
'\x4D',         /* M     */
'\x4E',         /* N     */
'\x4F',         /* O     */
'\x50',         /* P     */
'\x51',         /* Q     */
'\x52',         /* R     */
'\x53',         /* S     */
'\x54',         /* T     */
'\x55',         /* U     */
'\x56',         /* V     */
'\x57',         /* W     */
'\x58',         /* X     */
'\x59',         /* Y     */
'\x5A',         /* Z     */
'\x5B',         /* [     */
'\x5C',         /* \     */
```

```
'\x5D',         /* ]     */
'\x5E',         /* ^     */
'\x5F',         /* _     */
'\x60',         /* `     */
'\x61',         /* a     */
'\x62',         /* b     */
'\x63',         /* c     */
'\x64',         /* d     */
'\x65',         /* e     */
'\x66',         /* f     */
'\x67',         /* g     */
'\x68',         /* h     */
'\x69',         /* i     */
'\x6A',         /* j     */
'\x6B',         /* k     */
'\x6C',         /* l     */
'\x6D',         /* m     */
'\x6E',         /* n     */
'\x6F',         /* o     */
'\x70',         /* p     */
'\x71',         /* q     */
'\x72',         /* r     */
'\x73',         /* s     */
'\x74',         /* t     */
'\x75',         /* u     */
'\x76',         /* v     */
'\x77',         /* w     */
'\x78',         /* x     */
'\x79',         /* y     */
'\x7A',         /* z     */
'\x7B',         /* {     */
'\x7C',         /* |     */
'\x7D',         /* }     */
'\x7E',         /* ~     */
'\x7F',         /* DEL   */
'\x5F',         /*     80 */ /* Remap most of the C1/GR codes to _ */
'\x5F',         /*     81 */
'\x5F',         /*     82 */
'\x5F',         /*     83 */
'\x5F',         /* IND 84 */
'\x5F',         /* NEL 85 */
'\x5F',         /* SSA 86 */
'\x5F',         /* ESA 87 */
'\x5F',         /* HTS 88 */
'\x5F',         /* HTJ 89 */
'\x5F',         /* VTS 8A */
'\x5F',         /* PLD 8B */
'\x5F',         /* PLU 8C */
'\x5F',         /* RI  8D */
'\x5F',         /* SS2 8E */
'\x5F',         /* SS3 8F */
'\x5F',         /* DCS 90 */
```

```
'\x5F',         /* PU1 91 */
'\x5F',         /* PU2 92 */
'\x5F',         /* STS 93 */
'\x5F',         /* CCH 94 */
'\x5F',         /* MW  95 */
'\x5F',         /* SPA 96 */
'\x5F',         /* EPA 97 */
'\x5F',         /*     98 */
'\x5F',         /*     99 */
'\x5F',         /*     9A */
'\x5F',         /* CSI 9B */
'\x5F',         /* ST  9C */
'\x5F',         /* OSC 9D */
'\x5F',         /* PM  9E */
'\x5F',         /* APC 9F */
'\x5F',         /*     A0 */
'\x21',         /* ¡   A1 */
'\x63',         /* ¢   A2 */
'\x4C',         /* £   A3 */
'\x5F',         /*     A4 */
'\x59',         /* ¥   A5 */
'\x5F',         /*     A6 */
'\x5F',         /* §   A7 */
'\x4F',         /* ¨   A8 */
'\x5F',         /* ©   A9 */
'\x61',         /* ª   AA */
'\x5F',         /* «   AB */
'\x5F',         /*     AC */
'\x5F',         /*     AD */
'\x5F',         /*     AE */
'\x5F',         /*     AF */
'\x6F',         /* Ô   B0 */
'\x2B',         /* ±   B1 */
'\x32',         /* ²   B2 */
'\x33',         /* ³   B3 */
'\x5F',         /*     B4 */
'\x75',         /* µ   B5 */
'\x5F',         /* ¶   B6 */
'\x2E',         /* ·   B7 */
'\x5F',         /*     B8 */
'\x31',         /* ¹   B9 */
'\x6F',         /* º   BA */
'\x5F',         /* »   BB */
'\x5F',         /* ¼   BC */
'\x5F',         /* ½   BD */
'\x5F',         /*     BE */
'\x3F',         /* ¿   BF */
'\x41',         /* À   C0 */
'\x41',         /* Á   C1 */
'\x41',         /* Â   C2 */
'\x41',         /* Ã   C3 */
'\x41',         /* Ä   C4 */
```

```
'\x41',         /* Å   C5 */
'\x5F',         /* Æ   C6 */
'\x43',         /* Ç   C7 */
'\x45',         /* È   C8 */
'\x45',         /* É   C9 */
'\x45',         /* Ê   CA */
'\x45',         /* Ë   CB */
'\x49',         /* Ì   CC */
'\x49',         /* Í   CD */
'\x49',         /* Î   CE */
'\x49',         /* Ï   CF */
'\x5F',         /*     D0 */
'\x4E',         /* Ñ   D1 */
'\x4F',         /* ò   D2 */
'\x4F',         /* Ó   D3 */
'\x4F',         /* Ô   D4 */
'\x4F',         /* Õ   D5 */
'\x4F',         /* Ö   D6 */
'\x5F',         /* ×   D7 */
'\x4F',         /* Ø   D8 */
'\x55',         /* Ù   D9 */
'\x55',         /* Ú   DA */
'\x55',         /* Û   DB */
'\x55',         /* Ü   DC */
'\x59',         /* Ý   DD */
'\x5F',         /*     DE */
'\x5F',         /* ß   DF */
'\x61',         /* à   E0 */
'\x61',         /* á   E1 */
'\x61',         /* â   E2 */
'\x61',         /* ã   E3 */
'\x61',         /* ä   E4 */
'\x61',         /* å   E5 */
'\x5F',         /* æ   E6 */
'\x5F',         /* ç   E7 */
'\x65',         /* è   E8 */
'\x65',         /* é   E9 */
'\x65',         /* ê   EA */
'\x65',         /* ë   EB */
'\x69',         /* ì   EC */
'\x69',         /* í   ED */
'\x69',         /* î   EE */
'\x69',         /* ï   EF */
'\x5F',         /*     F0 */
'\x6E',         /* ñ   F1 */
'\x6F',         /* ò   F2 */
'\x6F',         /* ó   F3 */
'\x6F',         /* ô   F4 */
'\x6F',         /* õ   F5 */
'\x6F',         /* ö   F6 */
'\x5F',         /* ÷   F7 */
```

```
'\x6F',         /* ø   F8 */
'\x75',         /* ù   F9 */
'\x75',         /* ú   FA */
'\x75',         /* û   FB */
'\x75',         /* ü   FC */
'\x79',         /* ý   FD */
'\x5F',         /*     FE */
'\x5F',         /*     FF */
};

/*
*+
*       Define chrarcter that can be translated to uppercase by subtracting
* 32 from their value.  These are "a" through "{", "}", "à" through "ï", and
* "ñ" through "ý".
*-
*/
static unsigned int CASE_TABLE[8] = {0, 0, 0, 0x2FFFFFFE, 0, 0, 0, 0x3FFEFFFF};

/*
*+
*       Define those characters that cannot be printed.
*-
*/
static unsigned int CTRL_TABLE[8] = {0xFFFFFFFF, 0, 0, 0x80000000, 0xFFFFFFFF, 0, 0, 0};
```

B.3 LRDRIVER.COM

This section contains the LRDRIVER.COM command procedure, which compiles and links the LRDRIVER.C device driver.

```
$ SET NOON
$ SAVED_VFY = F$VERIFY("NO","NO")
$ ON CONTROL_Y THEN GOTO QUIT
$ SET VERIFY=(PROCEDURE,NOIMAGE)
$!
$! LRDRIVER.COM
$! This is the compile and link procedure for the example device driver
$! LRDRIVER.C.
$!
$! Usage:
$!
$!   @LRDRIVER [DEBUG]
$!
$!   P1          If specified as DEBUG then a version of the driver is built
$!               that facilitates debugging with the High Level Language System
$!               Debugger.
$!               The default is to build a normal version of the driver.
$!
$!'F$VERIFY("NO")'
$!
$ DEBUG_CC_OPT = ""
$ IF P1 .NES. ""
$ THEN
$    IF P1 .NES. "DEBUG" THEN EXIT %X14         ! SS$_BADPARAM
$    DEBUG_CC_OPT = "/DEBUG/NOOPTIMIZE/DEFINE=DEBUG"
$ ENDIF
$!
$ IF F$TRNLNM("SRC$") .EQS. "" THEN DEFINE/NOLOG SRC$ 'F$ENVIRONMENT("DEFAULT")'
$ IF F$TRNLNM("LIS$") .EQS. "" THEN DEFINE/NOLOG LIS$ 'F$ENVIRONMENT("DEFAULT")'
$ IF F$TRNLNM("OBJ$") .EQS. "" THEN DEFINE/NOLOG OBJ$ 'F$ENVIRONMENT("DEFAULT")'
$ IF F$TRNLNM("EXE$") .EQS. "" THEN DEFINE/NOLOG EXE$ 'F$ENVIRONMENT("DEFAULT")'
$ IF F$TRNLNM("MAP$") .EQS. "" THEN DEFINE/NOLOG MAP$ 'F$ENVIRONMENT("DEFAULT")'
$!
$ SET VERIFY=(PROCEDURE,NOIMAGE)
$!
$! Compile the driver
$!
$ CC/STANDARD=RELAXED_ANSI89/INSTRUCTION=NOFLOATING_POINT/EXTERN=STRICT-
        /POINTER_SIZE=32-
        'DEBUG_CC_OPT'-
        /LIS=LIS$:LRDRIVER/MACHINE_CODE-
        /OBJ=OBJ$:LRDRIVER-
                SRC$:LRDRIVER -
                +SYS$LIBRARY:SYS$LIB_C.TLB/LIBRARY
$!
$! Link the driver
$!
$ LINK/ALPHA/USERLIB=PROC/NATIVE_ONLY/BPAGE=14/SECTION/REPLACE-
        /NODEMAND_ZERO/NOTRACEBACK/SYSEXE/NOSYSSHR-
        /SHARE=EXE$:SYS$LRDRIVER.EXE-          ! Driver image
        /DSF=EXE$:SYS$LRDRIVER.DSF-            ! Debug symbol file
        /SYMBOL=EXE$:SYS$LRDRIVER.STB-         ! Symbol table
```

```
        /MAP=MAP$:SYS$LRDRIVER.MAP/FULL/CROSS - ! Map listing
                SYS$INPUT:/OPTIONS
!
!   Define symbol table for SDA using all global symbols, not just
!   universal ones
!
SYMBOL_TABLE=GLOBALS
!
!   This cluster is used to control the order of symbol resolution.  All
!   psects must be collected off of this cluster so that it generates
!   no image sections.
!
CLUSTER=VMSDRIVER,,,-
        !
        !   Start with the driver module
        !
        OBJ$:LRDRIVER.OBJ,-
        !
        !   Next process the private interfaces.  (Only include BUGCHECK_CODES if
        !   used by the driver module).  The /LIB qualifier causes the linker to
        !   resolve references in the driver module to DRIVER$INI_xxx routines
        !   (which are defined in the module DRIVER_TABLE_INIT).
        !
        SYS$LIBRARY:VMS$VOLATILE_PRIVATE_INTERFACES/INCLUDE=(BUGCHECK_CODES)/LIB,-
        !
        !   Explicitly include routines for the initialization section - there
        !   will be no outstanding references to cause this to happen when STARLET
        !   is searched automatically.
        !
        SYS$LIBRARY:STARLET/INCLUDE:(SYS$DRIVER_INIT,SYS$DOINIT)

!
!   Use the COLLECT statement to implicitly declare the NONPAGED_EXECUTE_PSECTS
!   cluster.  Mark the cluster with the RESIDENT attribute so that the image
!   section produced is nonpaged.  Collect only the code psect into the cluster.
!
COLLECT=NONPAGED_EXECUTE_PSECTS/ATTRIBUTES=RESIDENT,-
        $CODE$

!
!   Coerce the psect attributes on the different data psects to that they
!   all match.  This will force NONPAGED_READWRITE_PSECTS cluster to yield only
!   one image section.
!
PSECT_ATTR=$LINK$,WRT
PSECT_ATTR=$INITIAL$,WRT
PSECT_ATTR=$LITERAL$,NOPIC,NOSHR,WRT
PSECT_ATTR=$READONLY$,NOPIC,NOSHR,WRT
PSECT_ATTR=$$$105_PROLOGUE,NOPIC
PSECT_ATTR=$$$110_DATA,NOPIC
PSECT_ATTR=$$$115_LINKAGE,WRT
```

```
!
!   Use a COLLECT statement to implicitly declare the NONPAGED_DATA_PSECTS
!   cluster.  Mark the cluster with the RESIDENT attribute so that the image
!   section produced is nonpaged.  Collect all the data psects into the cluster.
!
COLLECT=NONPAGED_READWRITE_PSECTS/ATTRIBUTES=RESIDENT,-
        !
        !   Psect generated by BLISS modules
        !
        $PLIT$,-
        $INITIAL$,-
        $GLOBAL$,-
        $OWN$,-
        !
        !   Psects generated by DRIVER_TABLES
        !
        $$$105_PROLOGUE,-
        $$$110_DATA,-
        $$$115_LINKAGE,-
        !
        !   Standard Psects generated by all languages,
        !   including the high level language driver module
        !
        $BSS$,-
        $DATA$,-
        $LINK$,-
        $LITERAL$,-
        $READONLY$

!
!   Coerce the program section attributes for initialization code so
!   that code and data  will be combined into a single image section.
!
PSECT_ATTR=EXEC$INIT_CODE,NOSHR
```

```
!
!   Use a COLLECT statement to implicitly declare the INITIALIZATION_PSECTS
!   cluster.  Mark the cluster with the INITIALIZATION_CODE attribute so that the image
!   section produced is identified as INITIALCOD.
!
!   These program sections have special names so that when the linker sorts them
!   alphabetically they will fall in the order: initialization vector table, code,
!   linkage, build table vector.  The order in which they are collected does not affect
!   their order in the image section.
!
!   This is the only place where code and data should reside in the
!   same section.
!
!   NOTE: The linker will attach the fixup vectors to this cluster.  This is expected.
!
COLLECT=INITIALIZATION_PSECTS/ATTRIBUTES=INITIALIZATION_CODE,-
        EXEC$INIT_000,-
        EXEC$INIT_001,-
        EXEC$INIT_002,-
        EXEC$INIT_CODE,-
        EXEC$INIT_LINKAGE,-
        EXEC$INIT_SSTBL_000,-
        EXEC$INIT_SSTBL_001,-
        EXEC$INIT_SSTBL_002
$!
$QUIT: ! 'F$VERIFY(SAVED_VFY)'
$ EXIT $STATUS
```

C

Sample IOGEN Configuration Building Module (ICBM)

This appendix contains a sample IOGEN configuration building module (ICBM) and a sample command procedure for compiling and linking the sample ICBM.

C.1 ICBM Example

```
#pragma module MMOV$ICBM "V1"

/*
 *    MMOV$ICBM -- MultiMedia Configuration Building Module
 *
 *    Copyright © Digital Equipment Corporation, 1996 All Rights Reserved.
 *    Unpublished rights reserved under the copyright laws of the United States.
 *
 *    The software contained on this media is proprietary to and embodies the
 *    confidential technology of Digital Equipment Corporation.  Possession,
 *    use, duplication or dissemination of the software and media is
 *    authorized only pursuant to a valid written license from Digital
 *    Equipment Corporation.
 *
 *    RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure by the U.S.
 *    Government is subject to restrictions as set forth in Subparagraph
 *    (c)(1)(ii) of DFARS 252.227-7013, or in FAR 52.227-19, as applicable.
 *
 *    ------------------------------------------------------------------------
 *
 * ABSTRACT:
 *
 *  This is the IOGEN Configuration Building Module for the Multimedia Services
 *  for OpenVMS.  The primary function of this module is to load the video
 *  driver for the AV301/321 FullSupreme Video PCI module, and the J300 Sound
 *  and Motion TurboChannel module; and to load the sound driver for the
 *  Microsoft Windows Sound System ISA module.
 *
 *  This module is a shareable image invoked by IOGEN as part of system
 *  autoconfiguration.  It initializes an Autoconfiguration Bus Mapping table,
 *  passes it back to IOGEN, and contains routines that load the video driver
 *  for the TurboChannel or the PCI, and the audio driver for the ISA and EISA
```

```
 *  buses.
 *
 *  Note:  This ICBM example does not work on the DEC 2000 model 300 or
 *  model 500 systems.  These systems are also sometimes
 *  known as the AXP 150, or the DECpc 150.
 *
 * ENVIRONMENT:
 *
 *      Merged as a shareable image by IOGEN Autoconfiguration.  Called in
 *      EXEC mode at IPL 0.  This image must be installed as a known image
 *      ($ INSTALL ADD SYS$SHARE:MMOV$ICBM_07.EXE, for example) in order
 *      for AUTOCONFIGURE to activate it.  If it's being activated by a test
 *      image (as in debugging) it need not be installed.
 *
 *      The image name must include the system type and optionally the
 *      CPU type.  These are obtained using the DCL lexical function
 *      F$GETSYI, parameters "SYSTYPE" and "CPUTYPE".  The systype and
 *      CPU type are appended to the end of the image name following an
 *      underscore.  These values must be two hex digits each.  For example,
 *      on a DEC 3000 Model 300, systype is 07 and CPU type is 02, so the
 *      ICBM image name on this system is MMOV$ICBM_0702.EXE.  The CPU
 *      type is optional; the ICBM name could be MMOV$ICBM_07.EXE.  The
 *      systype and CPU type are used by AUTOCONFIGURE to form the image
 *      name.  The same image can be used on many different systems, as
 *      long as it is name appropriately.
 *
 */

/*
 *  Include files
 */
#include <adpdef.h>             /* Adapter control block        */
#include <busarraydef.h>        /* Bus array                    */
#include <crbdef.h>             /* Channel request block        */
#include <ctype.h>              /* Character type macros        */
#include <dcdef.h>              /* Adapter type codes           */
#include <descrip.h>            /* String descriptor definitions*/
#include <hwrpbdef.h>           /* HWRPB field definitions      */
#include <ioc_routines.h>       /* for ioc$node_data            */
#include <iocdef.h>             /* for IOC$K_EISA_IRQ           */
#include <iogendef.h>           /* IOGEN symbols and item codes */
#include <ssdef.h>              /* system error codes           */
#include <starlet.h>            /* system service prototypes    */
#include <string.h>             /* C string definitions         */
#include <stsdef.h>             /* status decoding macros       */
#include <vecdef.h>             /* interrupt vector symbols     */
```

```
/*
 *  External routines in IOGEN
 */
int     iogen$ac_select ();
int     iogen$assign_controller ();
int     iogen$log ();
int     sys$load_driver ();

/*
 *  External references to system globals
 *
 *  Define a pointer to the hardware RPB.  This is used to locate the
 *  EISA configuration pointer table, which in turn is used to determine
 *  if a device supported by this ICBM exists on the system.
 *
 *  The hardware RPB is a data structure used to pass information from
 *  the system console firmware to the operating system.  For more
 *  details, refer to the OpenVMS AXP Internals and Data Structures
 *  manual, published by Digital Press (ISBN 1-55558-120-X).
 *
 *  Use exe$gpl_hwrpb_l, the 32-bit pointer, instead of exe$gpq_hwrpb, a
 *  64-bit pointer.
 */
extern HWRPB *exe$gpl_hwrpb_l;

/*
 *  Other external definitions
 *
 *  These are IOGEN status codes.  They are exported as the addresses of
 *  data cells from the IOGEN shareable image.
 */
extern int IOGEN$_ICBM_OK;        /* Exported by IOGEN shareable image */
extern int IOGEN$_EXCLUDE;        /* Exported by IOGEN shareable image */

/*
 *  Routine prototypes
 */
int             iogen$icbm_init (ABM **abm);
static int      mmov$configure_bus (int handle, ADP *adp);
static int      mmov$configure_bus_eisa (int handle, ADP *adp);
static void     connect_the_driver (int handle, ADP *adp, int mct_index,
                                    BUSARRAYENTRY *ba);
static int      write_crb_reconnect (CRB *crb, BUSARRAYENTRY *ba);
```

```
/*
 *  Data structures used within this module.
 */
/*
 * The form of a SYS$LOAD_DRIVER itemlist entry.
 */
typedef struct itemlist
    {
    short int   buffer_size;
    short int   item_code;
    void        *buffer_address;
    short       *return_length;
    } ITEMLIST;

/*
 *  Autoconfigure bus mapping table.  This table includes an entry
 *  for each bus this ICBM knows how to configure.  Each entry
 *  in the table consists of an adapter type and an associated
 *  routine in the ICBM that configures devices on this adapter.
 *  Autoconfigure refers to this table as it scans the system ADP
 *  list, to determine if this ICBM needs to be called.
 *
 *  The ABM structure is defined in IOGENDEF.H.
 */
static ABM mmov_abm[] =
    {
    AT$_PCI,  mmov$configure_bus,
    AT$_TC,   mmov$configure_bus,
    AT$_ISA,  mmov$configure_bus,
    AT$_EISA, mmov$configure_bus_eisa,
    0,        0                          /* Zero signals end of table */
    };
```

```
/*
 *  Multimedia device configuration table.  This table includes the
 *  8-byte device ID string the ICBM uses to determine if a device
 *  exists on the system, the device name and the driver name that
 *  corresponds to the device, and the adapter type for the bus the
 *  the device is on.
 *
 *  The device ID is assumed to be no larger than 8 bytes, and if it's
 *  smaller than 8 bytes, the significant data is in the lower bytes
 *  and the upper bytes are zero.  This is done to simplify comparisons;
 *  the device ID is treated as a 64-bit integer and one comparison is
 *  done.
 *
 *  The form of the device ID differs for different buses.  Sometimes
 *  it differs for the same device.
 *
 *  For the AV301/AV321, the device ID is the contents of the 32-bit
 *  PCI configuration space PCI ID register.  The upper 32 bits are
 *  zero.
 *
 *  For the AV300-AA, the device ID is the first 8 bytes of the
 *  Turbochannel option ROM.  This is the ASCII string "AV300-AA".
 *
 *  For the Microsoft Sound Board, the device ID varies.  On ISA
 *  machines, the console command "add_sound" sets the device ID to
 *  the string "PCXBJ"; VMS copies only the first four bytes of this
 *  string to the bus array device ID field, resulting in "PCXB".
 *
 *  On EISA machines, the default ECU configuration file identifies
 *  the Microsoft Sound Board as "ISA2000".  The ICBM checks for this
 *  complete string.
 *
 *  "MSB" as a device ID for the Microsoft Sound Board is included in
 *  the configuration table because it was useful in debugging ISA
 *  configurations.
 *
 *  The complete list of devices supported by this ICBM is:
 *
 *      AV301 -- FullVideo Supreme video capture PCI card
 *      AV321 -- FullVideo Supreme JPEG video capture PCI card with
 *                      on-board JPEG compression/decompression
 *      AV300 -- Sound and Motion J300 video capture Turbochannel card
 *                      with on-board JPEG compression/decompression
 *                      (Note that this board also has sound capability,
 *                      which is not supported!)
 *      Microsoft Sound System Sound Card
 *      Oak Technologies Mozart Sound Card
 *      Sound card in Digital PCI workstations that are compatible
 *                      with the Microsoft Sound System.
 *
 *
 *  The configuration table definition is below.  The union for the device
```

```
 *  ID is necessary because the version of the DEC C compiler in use doesn't
 *  provide a way to statically initialize a 64-bit field.  Note that the
 *  low-order longword is first, then the high-order longword.
 */
static struct
    {
    union
        {
        struct
            {
            unsigned int id_l;
            unsigned int id_h;
            } id_fields;
        unsigned __int64 id;
        } u_id;
    char *device_name;
    char *driver_name;
    unsigned int adp_type;
    } config_table[] =

    {
   /*
    *  8-byte device ID    device     driver        adapter
    *   low       high      name       name
    *
    *   AV301 device ID
    */
    0x00131011, 0x00000000, "VI", "MMOV$VIDRIVER",  AT$_PCI,

    /*  AV321 device ID  */
    0x000E1011, 0x00000000, "VI", "MMOV$VIDRIVER",  AT$_PCI,

    /* "AV300-AA" in ASCII  */
    0x30335641, 0x41412D30, "VI", "MMOV$VIDRIVER",  AT$_TC,

    /*  "MSB" in ASCII */
    0x0042534D, 0x00000000, "AU", "MMOV$MSBDRIVER", AT$_ISA,

    /*  "PCXB" in ASCII */
    0x42584350, 0x00000000, "AU", "MMOV$MSBDRIVER", AT$_ISA,

    /*  "ISA2000" in ASCII */
    0x32415349, 0x00303030, "AU", "MMOV$MSBDRIVER", AT$_EISA
    };

static int config_table_size = sizeof(config_table) / sizeof(config_table[0]);
```

```
/*
 *  iogen$icbm_init -- IOGEN initialization routine for this ICBM
 *
 *  This routine is called by IOGEN when the ICBM is loaded.  It
 *  returns the address of the Autoconfiguration Bus Mapping Table (ABM).
 *
 *  Inputs:
 *
 *      abm -- address of a longword cell to receive the address
 *             of the ABM.
 *
 *  Outputs:
 *
 *      The address of the MMOV ABM is written to the input address.
 *
 *  Status Returns:
 *
 *      IOGEN$_ICBM_OK
 *
 *      Note:  the "&" operator is used below because this symbol is
 *      exported from the IOGEN shareable image as if it were the
 *      address of a data cell.
 */
int iogen$icbm_init(ABM **abm)
    {
    *abm = mmov_abm;
    return (int) &IOGEN$_ICBM_OK;
    }
```

```
/*
 *  mmov$configure_bus -- look for our devices in the ADP's bus array.
 *
 *  This routine scans the bus array pointed to by the input ADP for
 *  devices supported by this ICBM.  If a supported device is found, the
 *  associated driver is loaded and the unit connected.  If no device is
 *  found, exit.
 *
 *  Inputs:
 *
 *      handle -- magic number interpreted by the autoconfigure support
 *            routines
 *      adp -- pointer to the ADP to be checked for supported devices.
 *
 *  Implicit Inputs:
 *
 *      The multimedia device configuration table.
 *
 *  Outputs:
 *
 *      none
 *
 *  Implicit Outputs:
 *
 *      A device driver may be loaded, and a device unit created.
 *
 *  Status Returns:
 *
 *      SS$_NORMAL
 *
 *      The only errors that could be propagated up are those from
 *      the connect_the_driver routine, which does not return any errors.
 *
 */
static int mmov$configure_bus (int handle, ADP *adp)
    {
    int i, j, k;

    union
      {
      char    tempid1[8];
      __int64 tempid2;
      } u_tempid;

    char *id_ptr;

    BUSARRAY_HEADER *bah;
    BUSARRAYENTRY  *ba;

    bah = (BUSARRAY_HEADER *) adp->adp$ps_bus_array;
    ba = (BUSARRAYENTRY *) &bah->busarray$q_entry_list;
```

```
/*
 *  Scan through the bus array entry list looking for anything in
 *  the configuration table.  For each one found, load the driver and
 *  connect the unit.  All the bus differences, et al., are handled in
 *  the connect_the_driver routine.
 */
for (i = 0; i < bah->busarray$l_bus_node_cnt; i++)
    {
    /*
     *  The busarray$q_hw_id field is a 64-bit quantity.  Do a simple
     *  64-bit comparison to look for a match.  If this is an
     *  ISA ID, this was entered by the user at the console using the
     *  isacfg console command.  The user entered an up to 8-byte ASCII
     *  string.  Convert this string to upper case before doing the
     *  comparison, as a convenience to the user.
     */
    if (adp->adp$l_adptype == AT$_ISA)
        {
        id_ptr = (char *) &ba[i].busarray$q_hw_id;
        for (k = 0; k < 8; k++)
            {
            u_tempid.tempid1[k] = toupper(id_ptr[k]);
            }
        }
    else
        {
        u_tempid.tempid2 = ba[i].busarray$q_hw_id;
        }

    /*
     *  Compare the hardware ID in the bus array slot with each entry
     *  in the configuration table.  Load the driver on the first match
     *  in the configuration table.  As one last consistency check,
     *  make sure the adapter type in the configuration table matches
     *  that in the ADP.
     */
    for (j = 0; j < config_table_size; j++)
        {
        if ( u_tempid.tempid2 == config_table[j].u_id.id )
            {
            if (config_table[j].adp_type == adp->adp$l_adptype)
                {
                connect_the_driver(handle, adp, j,
                                   (BUSARRAYENTRY *) &ba[i]);
                break;
                }
            }
        }   /* for (j = 0; j < config_table_size; j++) */

    }  /* for (i = 0; i < bah->busarray$l_bus_node_cnt; i++) */

return (SS$_NORMAL);
}
```

```
/*
 *  mmov$configure_bus_eisa -- look for our devices in the console
 *                             data block
 *
 *      This routine locates our devices in the EISA Configuration
 *      Pointer Table.  This table is pointed to by the HWRPB offset
 *      HWRPB$IL_CDB_OFFSET_L.  This value is added to the base
 *      address of the HWRPB to get the address of the configuration
 *      pointer table.  The configuration pointer table consists
 *      of a collection of 3-longword items.  Each item contains
 *      a 7-byte ASCII identification string, followed by a 32-
 *      bit offset.  The offset is from the base of the Pointer
 *      Table, and points to the compressed configuration data
 *      block for that device.  The configuration data block, and
 *      the information in the pointer table, all come from data
 *      entered by the user using the ECU at the console prompt.
 *
 *      This is the only way to determine if an ISA device supported
 *      by this ICBM is configured on the system.
 *
 *      This routine is different from the mmov$configure_bus
 *      routine, because the location of the data we're looking
 *      for is very different.  The only thing in the bus array
 *      that's valid for an ISA device on an EISA bus is the CSR,
 *      and only the connect_the_driver routine cares about that.
 *
 *  NOTE NOTE NOTE NOTE NOTE
 *
 * This routine does not work on the DEC 2000 model 300 or model
 * 500 systems (also known as the AXP 150 or DECpc 150 systems).
 * The data structures describing the EISA bus on these systems
 * are laid out very differently than on subsequent
 * EISA-based systems, and this routine does not take these
 * differences into account.
 *
 *  Inputs:
 *
 *      handle -- magic number interpreted by the autoconfigure support
 *                routines
 *      adp -- pointer to the ADP
 *
 *  Implicit Inputs:
 *
 *      The multimedia device configuration table.
 *      The HWRPB, and the EISA configuration pointer
 *      table.
 *
 *  Outputs:
 *
 *      none
 *
 *  Implicit Outputs:
```

```
 *
 *      A device driver may be loaded, and a device unit created.
 *
 *  Status Returns:
 *
 *      SS$_NORMAL -- everything worked.
 *
 *      The only errors that could be propagated up are those from
 *      the connect_the_driver routine, which does not return any errors.
 *
 */
static int mmov$configure_bus_eisa (int handle, ADP *adp)
    {
    int found, i, j, status;

    HWRPB *hwrpb;

    BUSARRAY_HEADER *bah;
    BUSARRAYENTRY  *ba;

/*
 *  Define a structure for EISA configuration pointer table items.  Note
 *  that although the first 8 bytes are ASCII in the table, define it
 *  as an int64 here to make comparisons with the device configuration
 *  table easier.   Use the member_alignment pragma to make sure the
 *  compiler aligns the members on longword (4-byte) boundaries; otherwise
 *  the compiler inserts padding before the offset member that doesn't
 *  match how the data is laid out in memory.
 */
#pragma __member_alignment save
#pragma __nomember_alignment LONGWORD

typedef struct _cpt {
            __int64 device_id;
            void   *offset;
            } EISA_CPT;

#pragma __member_alignment restore

    EISA_CPT *cpt;

    /*
     *  Get the pointer to the bus array header, and a pointer to
     *  the first entry in the bus array.
     */
    bah = (BUSARRAY_HEADER *) adp->adp$ps_bus_array;
    ba = (BUSARRAYENTRY *) &bah->busarray$q_entry_list;

    /*
     *  Locate the configuration pointer table.  Note that the HWRPB cell
     *  is called "cdb_offset"; this cell points to the CPT, and the
     *  offset member of the CPT points to the CDB for each EISA slot.
     */
    hwrpb = exe$gpl_hwrpb_l;
    cpt = (EISA_CPT *) ( (char *)hwrpb + hwrpb->hwrpb$il_cdb_offset_l);
```

```
/*
 *  Scan through the CPT looking for our device.
 *
 *  This loop depends on the following facts:
 *
 *  1) the entries in the bus array and those in the CPT are in
 *     the same order.  This order is based on the hardware bus
 *     slot number.
 *  2) this order is maintained because the first entry (index zero)
 *     in the bus array describes the X-bus (the mouse, keyboard,
 *     floppy, etc. devices), and the first entry in the CPT is
 *     for the system motherboard (or CPU board).  The next entry
 *     in both the bus array and the CPT describes the first EISA
 *     slot.
 */
for (i = 0; i < bah->busarray$l_bus_node_cnt; i++)
    {
    /*
     *  Compare the hardware ID in the EISA configuration pointer table
     *  with each entry in the configuration table.  Load the driver on
     *  the first match in the configuration table.  As one last
     *  consistency check, make sure the adapter type in the configuration
     *  table matches that in the ADP.
     */
    for (j = 0; j < config_table_size; j++)
        {
        if ( cpt[i].device_id == config_table[j].u_id.id )
            {
            if (config_table[j].adp_type == adp->adp$l_adptype)
                {
                connect_the_driver(handle, adp, j,
                      (BUSARRAYENTRY *) &ba[i]);
                break;
                }
            }
        }  /* for (j = 0; j < config_table_size; j++) */

    }  /* for (i = 0; i < bah->busarray$l_bus_node_cnt; i++) */

return (SS$_NORMAL);
}
```

```
/*
 * connect_the_driver -- loads the driver and connects the unit
 *
 *  This routine loads the device driver specified by the configuration
 *  table entry, based on information in the ADP and the bus array
 *  entry.
 *
 *  Note that if the device is on an EISA bus, the IRQ is nowhere to be
 *  found in the ADP or bus array entry.  Unfortunately, this is needed to
 *  load the driver.  So, the code has to jump through a couple of extra
 *  hoops to find this value, as described below.
 *
 *  Inputs:
 *
 *      handle -- the autoconfigure magic number
 *      adp -- pointer to the ADP
 *      mct_index -- index into the multimedia configuration table
 *      ba -- pointer to the bus array entry for the device
 *
 *  Outputs:
 *
 *      none
 *
 *  Side Effects:
 *
 *      A device driver is loaded, and the required portions of the I/O
 *      database are created.  The driver's controller and unit
 *      initialization routines are executed.
 *
 *  Status Returns:
 *
 *      none.
 *
 */
static void connect_the_driver(int handle, ADP *adp, int mct_index, BUSARRAYENTRY *ba)
    {
    int status, temp_vector;
    char *device_name_pointer;
    short int iosb[4];
    ITEMLIST itemlist[6];               /* for LOAD_DRIVER */
    int arglist[4];                     /* for CMKRNL calls */
    CRB *crb;
    HWRPB *hwrpb;

    struct dsc$descriptor driver_name_desc;
    struct dsc$descriptor device_name_desc;
    char device_name[4];
```

```
/*
 *  Set up the driver name descriptor.
 */
driver_name_desc.dsc$w_length = strlen(config_table[mct_index].driver_name);
driver_name_desc.dsc$b_dtype = DSC$K_DTYPE_T;
driver_name_desc.dsc$b_class = DSC$K_CLASS_S;
driver_name_desc.dsc$a_pointer = config_table[mct_index].driver_name;

/*
 *  Construct the device name.  The first two letters are the device
 *  name from the configuration table, the next is the controller
 *  letter, which come from either the bus array entry or from an
 *  IOGEN routine, followed by an ASCII "0".
 */
 /*
  *  First fill in the device name descriptor.
  */
device_name_desc.dsc$w_length = 4;
device_name_desc.dsc$b_dtype = DSC$K_DTYPE_T;
device_name_desc.dsc$b_class = DSC$K_CLASS_S;
device_name_desc.dsc$a_pointer = &device_name[0];

/*
 *  Now, construct the device name proper, leaving out the
 *  controller letter.
 */
device_name_pointer = config_table[mct_index].device_name;
device_name[0] = device_name_pointer[0];
device_name[1] = device_name_pointer[1];
device_name[3] = '0';

/*
 *  If the system has assigned a controller letter already, use it.
 *  Otherwise, ask IOGEN to assign one.  iogen$assign_controller returns
 *  the assigned controller letter in busarray$b_ctrlltr.
 */
if (ba->busarray$b_ctrlltr == 0)
    {
    status = iogen$assign_controller (handle, device_name, ba);
    if (!$VMS_STATUS_SUCCESS(status))
        {
        device_name[2] = '?';
        iogen$log(handle, status, &device_name_desc, &driver_name_desc);
        return;
        }
   }
device_name[2] = ba->busarray$b_ctrlltr;
```

```
/*
 *  Check whether the device is being implicitly or explicitly excluded.
 *  AUTOCONFIGURE will return either SS$_NORMAL, to indicate the device
 *  should be configured; or IOGEN$_EXCLUDE, which means the device
 *  is implicitly or explicitly excluded.  In the latter case, simply
 *  return.
 */
status = iogen$ac_select(handle, &device_name_desc);
if (status == (int) &IOGEN$_EXCLUDE)  return;

/*
 *  Check to see this device is already configured.  If so, don't
 *  do it again -- multiple connects aren't allowed.  If the no_reconnect
 *  bit in the bus array entry flags is set, the device has already been
 *  connected.
 */
if (ba->busarray$v_no_reconnect)  return;

/*
 *  Now fill in the item list for $load_driver, specifying the adapter,
 *  the CSR, the interrupt vector, the node number, and a pointer to
 *  receive the address of the created CRB.  Like all standard VMS
 *  item lists, it is terminated by a zero.
 */

/*  The adapter   */
itemlist[0].buffer_size = sizeof(adp->adp$l_tr);
itemlist[0].item_code = IOGEN$_ADAPTER;
itemlist[0].buffer_address = &adp->adp$l_tr;
itemlist[0].return_length = 0;

/*  The CSR address */
itemlist[1].buffer_size = sizeof(ba->busarray$q_csr);
itemlist[1].item_code = IOGEN$_CSR;
itemlist[1].buffer_address = &ba->busarray$q_csr;
itemlist[1].return_length = 0;

/*
 *  The interrupt vector.  This is located in a different place for
 *  each bus, and is especially complicated for EISA.
 */
itemlist[2].buffer_size = sizeof(ba->busarray$l_bus_specific_l);
itemlist[2].item_code = IOGEN$_VECTOR;
itemlist[2].return_length = 0;
```

```
switch (adp->adp$l_adptype)
    {
    case AT$_PCI:
        /*
         *  If this is the PCI device, the interrupt vector comes
         *  from the low longword of the bus_specific field of the
         *  bus array entry.
         */
        itemlist[2].buffer_address = &ba->busarray$l_bus_specific_l;
        break;

    case AT$_TC:
        /*
         *  If it's Turbochannel, the vector comes from the
         *  autoconfig cell.
         */
        itemlist[2].buffer_address = &ba->busarray$l_autoconfig;
        break;

    case AT$_ISA:
        /*
         *  If this is ISA, the vector is the bus_specific_l
         *  cell times 4.
         */
        temp_vector = ba->busarray$l_bus_specific_l * 4;
        itemlist[2].buffer_address = &temp_vector;
        break;

    case AT$_EISA:
        /*
         *  If this is EISA, ask the system for the IRQ, via a call
         *  to IOC$NODE_DATA, which has to be called in kernel mode.
         *  This routine requires a CRB, which, unfortunately, isn't
         *  allocated 'til after the SYS$LOAD_DRIVER call.  So,
         *  allocate a private, local CRB, and fill in the values
         *  needed by IOC$NODE_DATA, namely CRB$L_NODE (which comes
         *  from BUSARRAY$L_NODE_NUMBER), and the address of the ADP.
         *  Don't worry about the rest of the local CRB contents, since
         *  the code path thru IOC$NODE_DATA doesn't touch anything else.
         *
         *  When IOC$NODE_DATA returns, multiply the IRQ times 4.
         */
        {
        CRB local_crb;
        VEC *vec;

        local_crb.crb$l_node = ba->busarray$l_node_number;
        vec = (VEC *) &local_crb.crb$l_intd;
        vec->vec$ps_adp = adp;
```

```
            arglist[0] = 3;
            arglist[1] = (int) &local_crb;
            arglist[2] = IOC$K_EISA_IRQ;
            arglist[3] = (int) &temp_vector;
            status = sys$cmkrnl (ioc$node_data, arglist);

            /*
             *  On certain platforms, 7 must be added to the IRQ to
             *  account for a hardware oddity in the way interrupt
             *  lines are wired.
             */

#ifndef HWRPB_SYSTYPE$K_LYNX  /* defined after V6.2 shipped */
#define HWRPB_SYSTYPE$K_LYNX 24
#endif
            hwrpb = (HWRPB *) exe$gpl_hwrpb_l;

            switch (hwrpb->hwrpb$iq_systype)
                {
                case HWRPB_SYSTYPE$K_TURBOLASER:  /* Alphaserver 8200/8400 */
                case HWRPB_SYSTYPE$K_SABLE:       /* Alphaserver 2100      */
                case HWRPB_SYSTYPE$K_LYNX:        /* Alphaserver 2100A     */
                    temp_vector += 7;
                    break;

                default:
                    break;
                }

            temp_vector *= 4;
            itemlist[2].buffer_address = &temp_vector;
            break;
            }

        default:
            /*
             *  If control gets to here, it's an adapter type this
             *  routine doesn't know how to handle.  Simply return.
             */
            return;
            break;
        }  /*  switch (adp->adp$l_adptype) */

    /*  The node number */
    itemlist[3].buffer_size = sizeof(ba->busarray$l_node_number);
    itemlist[3].item_code = IOGEN$_NODE;
    itemlist[3].buffer_address = &ba->busarray$l_node_number;
    itemlist[3].return_length = 0;

    /*  A pointer to the CRB to be allocated  */
    itemlist[4].buffer_size = sizeof(crb);
    itemlist[4].item_code = IOGEN$_CRB;
    itemlist[4].buffer_address = &crb;
    itemlist[4].return_length = 0;
```

```
/*  Terminate the list */
itemlist[5].buffer_size = 0;
itemlist[5].item_code = 0;
itemlist[5].buffer_address = 0;
itemlist[5].return_length = 0;

status = sys$load_driver(IOGEN$_CONNECT, &device_name_desc,
                         &driver_name_desc, itemlist, iosb);

if ($VMS_STATUS_SUCCESS(status)) status = (int) iosb[0];

/*
 *  Log errors here through IOGEN's logging function.  This
 *  information is output when the /LOG qualifier is used on
 *  the AUTOCONFIGURE command.
 */
iogen$log(handle, status, &device_name_desc, &driver_name_desc);

/*
 * sys$load_driver can return two errors which can be ignored:
 * SS$_DEVEXISTS and SS$_DEVOFFLINE.  DEVEXISTS can never happen
 * if the no_reconnect bit is used. Logic above uses this bit to
 * prevent attempting to load the driver twice.  DEVOFFLINE means
 * that for some reason, the device didn't come on line by the
 * time sys$load_driver completed its work.  This could be because
 * some units take a long time to come on line.  Or it could be
 * because the driver detected some error during controller or
 * unit initialization -- the multimedia video and audio drivers
 * work this way.  Either way, these are not errors as far as
 * the ICBM is concerned, so they are ignored.
 *
 * On any other error, exit here without changing the state of the
 * no_reconnect bit.
 */
if ( (status == SS$_DEVEXISTS) || (status == SS$_DEVOFFLINE))
    status = SS$_NORMAL;
if (!$VMS_STATUS_SUCCESS(status)) return;
```

```
    /*
     * Now that the driver is loaded and the unit connected, save the
     * address of the CRB in the bus array entry.  Also, set the
     * no_reconnect flag to indicate the unit cannot be re-connected.
     * These operations must be done in kernel mode, because the memory
     * containing the bus array entry is not writeable in exec mode.
     *
     * Normally, these two functions would be perfomed by a callback to
     * IOGEN.  However, the IOGEN shareable image does not export these
     * functions.  So, there is one little kernel-mode routine to do
     * the work.  Finally, note that the status from the sys$cmkrnl call
     * is ignored.  It's there as a debugging aid.
     */
    arglist[0] = 2;
    arglist[1] = (int) crb;
    arglist[2] = (int) ba;
    status = sys$cmkrnl (write_crb_reconnect, arglist);

    return;
    }

/*
 * write_crb_reconnect -- write the CRB address and set the no_reconnect bit
 *
 *  This routine duplicates the actions of the kernel mode IOGEN routines
 *  IOGEN$WRITE_IOGEN_CRB and IOGEN$SET_NORECONNECT, which are not exported
 *  by the IOGEN shareable image.  The address of the specified CRB is written
 *  into busarray$ps_crb, and the no_reconnect bit is set.
 *
 *  This routine must be called in kernel mode, because the memory containing
 *  the bus array entry is not writeable in exec mode.
 *
 *  Inputs:
 *
 *      crb -- address of the CRB to write into the bus array entry
 *      ba  -- pointer to the bus array entry
 *
 *  Outputs:
 *
 *      none
 *
 *  Side Effects:
 *
 *      none
 *
 *  Completion Codes:
 *
 *      SS$_NORMAL
 *
 */
static int write_crb_reconnect (CRB *crb, BUSARRAYENTRY *ba)
    {
```

```
    ba->busarray$ps_crb = crb;
    ba->busarray$v_no_reconnect = 1;
    return SS$_NORMAL;
    }
```

C.2 ICBM Example Command Procedure

This section contains the SYS$ICBM_EXAMPLE.COM command procedure, which compiles and links the example ICBM SYS$ICBM_EXAMPLE.C.

```
$ SET NOON
$ SAVED_VFY = F$VERIFY("NO","NO")
$ ON CONTROL_Y THEN GOTO QUIT
$ SET VERIFY=(PROCEDURE,NOIMAGE)
$!
$!  SYS$ICBM_EXAMPLE.COM
$!
$!  This procedure builds the example ICBM SYS$ICBM_EXAMPLE.C, creating the
$!  image SYS$ICBM_EXAMPLE.EXE.
$!
$!  This procedure assumes that the source file is in the SRC$ directory,
$!  the object and executable file will be put into the OBJ$ directory, and
$!  that listings and link maps will be put into the LIS$ directory.  If
$!  these logical names are not defined, this procedure defines them to be
$!  the local directory.
$!
$!  This procedure assumes the DEC C compiler is installed, and is the
$!  default compiler run with the "CC" DCL command.  The example ICBM is
$!  written in C, and requires the DEC C compiler.
$!
$!  Usage:
$!
$!    @SYS$ICBM_EXAMPLE [DEBUG]
$!
$!    P1        If specified as DEBUG then the ICBM is built with the
$!              debugger.  Note that since the ICBM code runs in EXEC
$!              mode, the DELTA debugger must be used.  To use this
$!              debugger, first define the logical name LIB$DEBUG to
$!
$!
$!
$ DEBUG_CC_OPT = ""
$ IF P1 .NES. ""
$ THEN
$    IF P1 .NES. "DEBUG" THEN EXIT %X14
$    DEBUG_CC_OPT = "/DEBUG/NOOPTIMIZE"
$ ENDIF
$!
$ IF F$TRNLNM("SRC$") .EQS. "" THEN DEFINE/NOLOG SRC$ 'F$ENVIRONMENT("DEFAULT")'
$ IF F$TRNLNM("LIS$") .EQS. "" THEN DEFINE/NOLOG LIS$ 'F$ENVIRONMENT("DEFAULT")'
$ IF F$TRNLNM("OBJ$") .EQS. "" THEN DEFINE/NOLOG OBJ$ 'F$ENVIRONMENT("DEFAULT")'
```

```
$ IF F$TRNLNM("EXE$") .EQS. "" THEN DEFINE/NOLOG EXE$ 'F$ENVIRONMENT("DEFAULT")'
$ IF F$TRNLNM("MAP$") .EQS. "" THEN DEFINE/NOLOG MAP$ 'F$ENVIRONMENT("DEFAULT")'
$!
$ SET VERIFY=(PROCEDURE,NOIMAGE)
$!
$!  Compile the ICBM
$!
$ CC 'DEBUG_CC_OPT'/OBJECT=OBJ$:/LIST=LIS$:/MACHINE/INSTRUCTION=NOFLOAT/EXTERN=STRICT-
        /L_DOUBLE=64 /STANDARD=RELAXED /show=include -
        SRC$:SYS$ICBM_EXAMPLE.C + SYS$LIBRARY:SYS$LIB_C.TLB/LIBRARY
$!
$!  Link the ICBM
$!
$ LINK /ALPHA/USERLIB=PROC/NATIVE_ONLY/BPAGE/SECTION/REPLACE/VMS_EXEC -
       /NODEMAND_ZERO/SYSEXE/NOSYSSHR/NOTRACEBACK -
       /MAP=MAP$:SYS$ICBM_EXAMPLE.MAP/FULL/CROSS -
       /SHARE=EXE$:SYS$ICBM_EXAMPLE.EXE -
       SYS$INPUT:/OPTIONS
       OBJ$:SYS$ICBM_EXAMPLE.OBJ,-
       sys$library:starlet.olb/include=(iogen$icbm_control),-
       sys$share:iogen$share/share
$!
$QUIT:
$ EXIT $STATUS
```

Index

C

D

E

F

H

I

J

K

L

M

N

O

P

Q

R

S

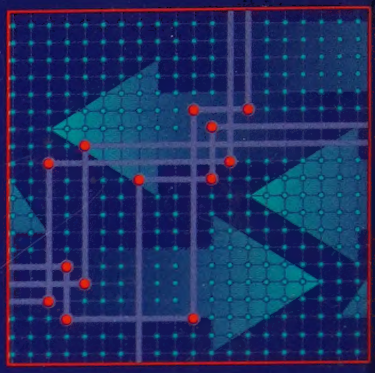
T

U

V

W

Computing / C Programming



Writing OpenVMS Alpha Device Drivers in C

DEVELOPER'S GUIDE AND REFERENCE MANUAL

Margie Sherlock • Leonard Szubowicz

Writing OpenVMS Alpha Device Drivers in C provides definitive information about writing device drivers in the C programming language for a device connected to an Alpha processor.

The book introduces the components of OpenVMS Alpha device drivers and explains their role in the operating system. Detailed chapters cover how to code, compile, and link drivers and how to load them into the operating system. An expanded reference section defines data structures, routines, and macros used in OpenVMS Alpha device driver programming.

Throughout ***Writing OpenVMS Alpha Device Drivers in C,*** examples of actual code illustrate the key concepts. The working sample device driver demonstrates an actual OpenVMS Alpha device driver written in C.

Margie Sherlock, a technical principal writer with Digital Equipment Corporation, has been writing about the OpenVMS operating system for over eight years. She has documented many components of the OpenVMS VAX and Alpha operating systems. Margie is the author of ***Using DECwindows Motif for OpenVMS,*** also published by Digital Press.

Leonard Szubowicz, a consulting software engineer with Digital Equipment Corporation, has been a contributor to the OpenVMS operating system for over nine years. As the architect and project leader for the High-Level Language Device Driver development effort, he was responsible for designing and implementing I/O support for writing OpenVMS Alpha drivers in C.

ISBN 1-55558-133-1

90000

9 781555 581336

dp Digital Press
An imprint of Butterworth-Heinemann

EY-T133E-DP